La Differentiation automatique de fonctions representees par des programmes

Jean Charles Gilbert, Georges Le Vey, J. Masse

Rapports de Recherche

N°1557

Programme 5

Traitement du signal,
Automatique et Productique

LA DIFFERENTIATION AUTOMATIQUE DE FONCTIONS REPRESENTEES PAR DES PROGRAMMES

Jean Charles GILBERT
Georges LE VEY
John MASSE

Novembre 1991

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

LA DIFFERENTIATION AUTOMATIQUE DE FONCTIONS REPRESENTEES PAR DES PROGRAMMES

AUTOMATIC DIFFERENTIATION OF FUNCTIONS REPRESENTED BY PROGRAMS

Jean Charles GILBERT, Georges LE VEY et John MASSE

Rapport de Recherche n° 1557

Novembre 1991

LA DIFFERENTIATION AUTOMATIQUE DE FONCTIONS REPRESENTEES PAR DES PROGRAMMES*

AUTOMATIC DIFFERENTIATION OF FUNCTIONS REPRESENTED BY PROGRAMS*

Jean Charles GILBERT†, Georges LE VEY‡ et John MASSE‡

Résumé

La *différentiation automatique* est un ensemble de méthodes permettant de générer de manière automatique un programme calculant les dérivées d'une fonction à partir d'un programme calculant la fonction elle-même. Avec la méthode connue sous le nom de *mode inverse*, on peut générer un programme calculant la valeur du *gradient* d'une fonction scalaire en un point, en un temps *relatif* indépendant du nombre de variables dont elle dépend. Ce temps ne dépasse pas un faible nombre de fois celui nécessaire au calcul de la valeur de la fonction.

Ce rapport fait le point sur les différentes méthodes de différentiation automatique, sur leurs techniques d'implémentation et sur leur utilisation dans l'analyse de la sensibilité des programmes aux erreurs d'arrondi. Des essais comparatifs avec les principaux codes actuellement disponibles ont été réalisés, couvrant notamment le cas du calcul de gradients, de jacobiennes et de la dérivée de fonctions définies au moyen d'un processus itératif.

Abstract

Automatic differentiation is a set of methods allowing to generate automatically a program computing the derivatives of a function from a program computing the function itself. With the method known as the *reverse mode*, it is possible to generate a program computing the value of the *gradient* of a scalar function at a point, in a *relative* time independent of the number of variables the function depends on. This time does not exceed a few times the one required to compute the value of the function.

This report reviews the methods of automatic differentiation, their implementation techniques and how to use them in the analysis of the sensitivity of programs to rounding. Comparative tests have been made, covering among others the computation of gradients, of Jacobians and of derivatives of a function defined by an iterative process.

Mots-clés: Différences divisées, différentiation automatique, différentiation de programmes, différentiation symbolique, erreurs d'arrondi, état adjoint, sensibilité.

---

* Travail supporté en partie par le Centre National d'Etudes Spaciales (CNES), 3 avenue Edouard Belin, F-31055 Toulouse.

† INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, F-78153 Le Chesnay. Tél: 33/1/39.63.55.24. Cél: gilbert@fermat.inria.fr.

‡ SIMULOG, 1 rue James Joule, F-78182 Saint Quentin en Yvelines.

Table des matières

1 Introduction

Autant le préciser dès le début: la *différentiation automatique* de (fonctions représentées par des) programmes n'est pas ce qu'il est convenu d'appeler la *différentiation symbolique* telle qu'on la trouve dans des codes comme MACSYMA [68], MAPLE [7] ou REDUCE. Celle-ci suppose que la fonction que l'on cherche à dériver admette une représentation symbolique au moyen de formules mathématiques. Elle obtient alors une représentation, également symbolique, de la fonction dérivée. Ce point de vue peut être vu comme une automatisation de ce qu'un être humain a appris à faire sur une feuille de papier. Notons que des codes comme MACSYMA, MAPLE ou REDUCE vont bien au delà de la différentiation de formules mathématiques et offrent toute une panoplie de manipulations symboliques (ou *calcul formel*) que ne vise pas à réaliser la différentiation automatique.

De son côté, la différentiation automatique ne suppose pas que la fonction à dériver doive pouvoir être représentée symboliquement mais suppose qu'une représentation au moyen d'un programme informatique soit possible et disponible. Dans un sens, ceci est plus général. Par exemple, il est en principe permis que la fonction soit définie au moyen d'instructions conditionnelles, de boucles répétitives, d'appels (éventuellement récursifs) à des sous-routines, de processus itératifs, etc .... Toutes sortes de choses qui n'ont pas d'équivalent symbolique. Toutefois, notons que cette représentation par programme se situe à un niveau moins structuré que la représentation symbolique et ceci ne sera pas sans conséquences.

Mais la différentiation automatique n'est pas non plus de la *différentiation par différences divisées* (ou *différences finies*), méthode par laquelle les dérivées partielles de la fonction considérée sont estimées en faisant varier insensiblement et successivement chacune des variables, tout en gardant les autres fixées. La différence essentielle entre les deux méthodes vient du fait que les dérivées obtenues par différentiation automatique sont exactes aux erreurs d'arrondi près. Il n'y a donc pas d'approximation sur les formules donnant les dérivées, comme c'est le cas pour les différences divisées.

Si la différentiation automatique accepte en entrée un programme, c'est aussi un programme qui représentera la fonction dérivée. Il ne sera en général pas question d'essayer d'obtenir à partir de celui-ci une représentation visuelle ou symbolique de la fonction dérivée. Ceci serait d'une difficulté extrême, sinon impossible. Par contre, ce qui sera disponible c'est la *valeur* de la dérivée en un point donné. Il suffira pour l'obtenir de faire 'tourner' le programme dérivé. Nous verrons en section 2 deux concepts permettant de générer de tels programmes. La mise en oeuvre informatique de ces concepts sera examinée en section 4. Elle pourra se faire soit par *précompilation*, soit par la technique de *surcharge des opérateurs* que l'on trouve dans les langages de programmation tels que C++, ADA ou FORTRAN-8X.

Insistons bien sur le point suivant. Si la différentiation automatique est une technique concurrente de la différentiation symbolique, il ne faut pas la voir comme cherchant à la supplanter dans son domaine. Nous pensons en effet que les deux approches sont complémentaires et ont des domaines de prédilection différents. Ainsi la différentiation symbolique, jointe à la génération automatique de code en langage de haut niveau, a un attrait indéniable puisqu'elle permet une approche voisine du travail symbolique coutumier et libère l'utilisateur de l'écriture d'un programme. D'un autre côté, certaines fonctions se présentent naturellement sous forme de programmes sans représentation symbolique avantageuse. Par exemple, dans les tableurs comme EXCEL (Microsoft) l'utilisateur fait dépendre certaines cases de sa feuille de calcul d'autres cases par des chemins et opérations multiples. L'évaluation de la

'sensibilité' d'une case par rapport à une autre revient à un calcul de dérivée. Or, dans ce cas, devoir construire une représentation symbolique des fonctions ainsi construites serait une tâche fastidieuse et coûteuse. Ici, c'est le cadre de la différentiation automatique qui s'avère être le mieux adapté à la situation.

Nous sentons qu'il est nécessaire de mettre en avant dès à présent un des atouts majeurs de la différentiation automatique que la différentiation symbolique ne peut acquérir que dans certaines circonstances. Pour cela, considérons le cas où l'on cherche à calculer les dérivées partielles premières d'une fonction (scalaire) f dépendant de n variables. On sait que lorsque le nombre de variables devient grand, l'efficacité en temps de calcul du code généré par différentiation symbolique a tendance à se détériorer. En effet, si l'on se passe des techniques lourdes d'optimisation de code, le temps de calcul est de l'ordre de n fois le temps nécessaire au calcul de la fonction. Or, pour le même type de fonctions, la différentiation automatique a un mode de fonctionnement, dit *mode inverse*, avec lequel le temps nécessaire au calcul du gradient rapporté au temps de calcul de f est indépendant de n. Cet avantage est très important.

En fait, en allant un peu plus en avant dans la discussion, on peut s'apercevoir que différentiations symbolique et automatique ne sont pas si éloignées l'une de l'autre. Nous connaissons en effet un cas où cette règle sur le temps de calcul par différentiation symbolique peut être contournée. C'est le cas de certains problèmes de contrôle optimal pour lesquels la différentiation symbolique peut être utilisée conjointement avec la génération automatique d'un état adjoint (voir par exemple [46, 47] pour un code développé sous MACSYMA). Il se fait que l'on est alors très proche du mode inverse de la différentiation automatique. En effet, dans ce mode, c'est également un état adjoint qui est introduit mais au lieu d'être associé aux équations d'état, il est associé aux lignes du programme représentant la fonction, ces lignes jouant le rôle de contraintes.

Un autre atout du mode inverse de la différentiation automatique est de fournir, sans coût supplémentaire, des quantités mesurant la sensibilité d'une variable spécifiée (par exemple, une des composantes de la fonction que l'on veut dériver) par rapport à *toutes* les autres variables du code jouant un rôle dans le calcul de la variable en question. Cette information peut être utilisée à diverses fins, comme au repérage des opérations contribuant le plus à la détérioration de la précision des calculs.

Avec ce rapport, nous poursuivons deux objectifs. D'une part, nous cherchons à recenser, décrire et analyser les principes de différentiation automatique (section 2), ainsi que les techniques de leur mise en oeuvre (section 4). Ceci nous amènera à présenter (section 5) et à tester (section 6) les principaux codes actuellement disponibles. D'autre part, nous examinerons les possibilités d'utilisation de ces techniques pour l'étude de la propagation des erreurs d'arrondi dans les programmes de calcul numérique (section 3).

Le style de la première partie de ce rapport (sections 2–4) est résolument du type "état de l'art". Notre parti est de présenter un point de vue unificateur sur les résultats et solutions existants, pouvant servir de base à des développements futurs. Cependant, quelques résultats nouveaux ont été glissés ici et là dans le texte. Ils concernent notamment la différentiation automatique de fonctions représentées par des processus itératifs, la complexité du calcul de la diagonale d'une jacobienne et, bien sûr, les résultats comparatifs de la section 6.

Nous tenons à remercier Andreas Griewank pour les discussions fructueuses que nous

avons eues et pour son acceuil à l'Argonne National Laboratory. Beaucoup de ses "communications personnelles" ont permis d'améliorer ce rapport sur divers points.

2 Principes des méthodes

2.1 Les fonctions concernées

Commençons par considérer le cas d'une fonction scalaire réelle

$$f : I\!R^n \longrightarrow I\!R$$

de n variables réelles $x_1, \ldots, x_n$, que nous appellerons *variables indépendantes*. Le cas de fonctions à valeurs vectorielles sera examiné en section 2.4. Comme nous l'avons dit dans l'introduction, l'hypothèse de base de la différentiation automatique est que cette fonction est connue par la donnée d'un programme de calcul qui en un point $x = (x_1, \ldots, x_n)$ de $I\!R^n$ fournit la valeur $f(x)$. On cherche alors à générer un programme qui en un point $x \in I\!R^n$ donné calculera la valeur numérique $\nabla f(x)$ du gradient de f en x, c'est-à-dire l'ensemble de ses dérivées partielles premières $\partial f/\partial x_1(x), \ldots, \partial f/\partial x_n(x)$.

Idéalement, un code de différentiation automatique devrait pouvoir prendre en compte toute fonction f qui peut être calculée au moyen d'un programme. C'est une exigence très forte. En particulier, cela implique que le différentiateur doit pouvoir traiter des fonctions qui ne sont pas nécessairement représentables par des formules mathématiques explicites. Par exemple, une fonction implicite dont le calcul de la valeur demanderait la résolution d'un système d'équations non linéaires par une méthode itérative. La structure informatique du programme peut aussi contenir des instructions de contrôle (`if-then-else`, `goto`, ...), des boucles répétitives, des sous-routines, des processus itératifs et récursifs, etc. On comprend qu'un programme calculant une fonction peut être très complexe.

Le plus sûr moyen de faire face à cette complexité est de procéder par étapes et de ne considérer, dans un premier temps, que des programmes ayant une structure simple. Un code de différentiation automatique est en effet basé sur une méthodologie dictant les actions à prendre en chaque circonstance. Cette méthodologie repose elle-même sur un modèle de programmes. Plus le modèle est général, plus le différentiateur admet des instructions variées sans avoir à transformer le programme original.

On rencontre fréquemment deux types de modèles que nous appellerons *modèle simple* et *modèle étendu*. Le premier est moins général que le second dans le sens où il demande plus de transformations du programme à "différentier". Tous deux supposent que celui-ci est formé d'une suite de K instructions d'affectation, en particulier, qu'il est exempt d'instructions de contrôle, de boucles répétitives, de sous-routines, etc. Nous verrons en section 4 comment prendre en compte de telles instructions.

Autant que possible nous nous placerons dans le cadre du modèle étendu. Toutefois, certaines méthodes et certains points de vue ne seront valables et ne pouront s'exprimer que dans le cadre du modèle simple.

2.1.1 Le modèle simple

Dans le cas le plus simple, un programme calculant une fonction f de n variables indépendantes $x_1, \ldots, x_n$, le fait en utilisant un certain nombre de variables auxiliaires $x_{n+1}, \ldots, x_N$

($N > n$), que nous appellerons *variables intermédiaires*. Si le programme est consistant, chaque fois qu'une nouvelle variable intermédiaire x_i est introduite, sa valeur est calculée à partir d'autres variables dont la valeur doit être connue au moment du calcul de x_i. En adoptant une numérotation convenable des variables, on peut supposer que la variable introduite x_i a un numéro d'ordre supérieur à celui des variables déjà calculées. Par conséquent, la première fois que x_i recevra une valeur, ce sera au moyen d'une intruction d'affectation de la forme:

$$x_i := \varphi_i(x_1, \ldots, x_{i-1}),$$

où φ_i ne dépend que de $x_1, \ldots, x_{i-1}$. Les fonctions φ_i sont appelées *fonctions intermédiaires*.

L'hypothèse de base du *modèle simple*[1] est qu'une nouvelle variable est introduite à *chaque* instruction d'affectation. Autrement dit, un programme obéissant à ce modèle pourra être schématisé (dans un langage qui nous est propre) de la manière suivante:

$$\left[\begin{array}{l} \textbf{for} \quad i := n+1 \quad \textbf{to} \quad N \quad \textbf{do} \quad x_i := \varphi_i(x_1, \ldots, x_{i-1}); \\ f := x_N. \end{array}\right.$$

Ce programme manipule donc N variables (n variables indépendantes et $N - n$ variables intermédiaires) et la valeur de $f(x)$ est récupérée dans la variable x_N. Il y a $K = N - n$ instructions d'affectation.

En général, les fonctions intermédiaires φ_i ne dépendent que d'une partie des variables calculées précédemment: il existe une partie P_i de $\{1, \ldots, i-1\}$ telle que $x_i = \varphi_i(x_{P_i})$. On note $x_{P_i} = \{x_j : j \in P_i\}$. Avec ces notations, l'algorithme précédent peut se récrire:

$$\left[\begin{array}{l} \textbf{for} \quad i := n+1 \quad \textbf{to} \quad N \quad \textbf{do} \quad x_i := \varphi_i(x_{P_i}); \\ f := x_N. \end{array}\right. \tag{1}$$

Le modèle simple manque de généralité sur au moins deux points. D'abord, un programme peut contenir des instructions de contrôle qui font qu'il n'est pas possible de déterminer *a priori* l'ordre dans lequel les variables intermédiaires seront introduites. Cet ordre peut dépendre, en effet, des valeurs données aux variables indépendantes. Toutefois, si le programme est bien conçu et si l'on donne les valeurs $x_1^0, \ldots, x_n^0$ aux variables indépendantes $x_1, \ldots, x_n$, la règle sur l'introduction des variables intermédiaires énoncées ci-dessus pourra être respectée pour autant que l'on adapte la numérotation de ces variables à l'exécution particulière considérée. De plus, il est raisonnable de supposer que cette numérotation restera admissible pour des valeurs des variables indépendantes voisines de $x_1^0, \ldots, x_n^0$. Les différentiateurs automatiques basés sur le modèle simple de programme utilisent une numérotation dépendant de l'exécution.

Le modèle simple est restrictif pour une autre raison. Il est en effet fréquent de trouver dans les programmes des variables intermédiaires qui sont redéfinies, c'est-à-dire des variables qui reçoivent des valeurs différentes en divers endroits du programme. Cela n'est pas permis dans le modèle simple. Dès lors, les algorithmes de différentiation automatique basés sur ce modèle se voient forcés de transformer le programme original en introduisant des variables intermédiaires supplémentaires à chaque redéfinition de variables. Dans certains cas, ceci peut constituer un accroissement appréciable de l'espace mémoire requis (que l'on songe par

[1] En anglais, on parle de fonctions représentables par *code list* (Rall (1981)), de *straight-line algorithms* (Miller) ou de *feasible algorithms* (Kim et al. (1984)).

exemple au cas de boucles `do` emboîtées dans lesquelles des variables intermédiaires sont redéfinies).

Un programme obéissant au modèle simple admet une représentation sous forme de *graphe de calcul* (*computational graph*) ou *graphe de Kantorovitch* [36]. Chaque noeud du graphe représente une variable (indépendante ou intermédiaire) et un arc allant du noeud x_j au noeud x_i signifie que $j \in P_i$: voir figure 1. L'ensemble P_i est formés des indices "précédant"

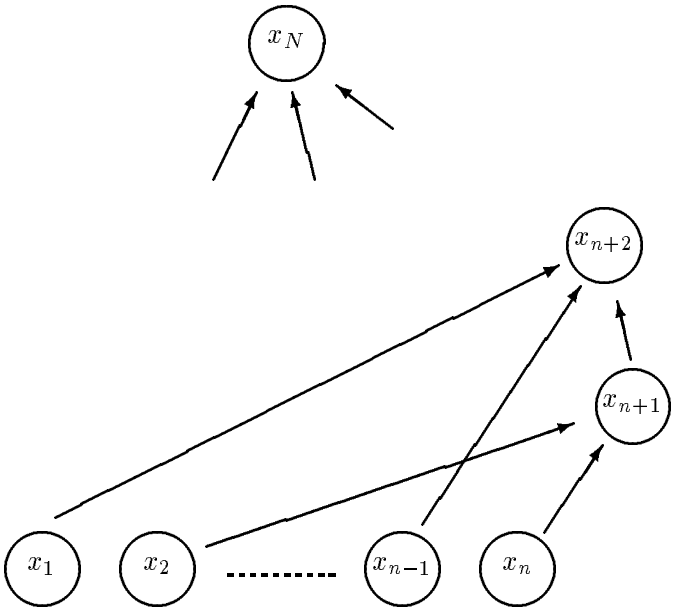

Figure 1: Graphe de calcul d'une fonction.

l'indice i. Il est également intéressant de considérer l'ensemble S_j des indices "succédant" l'indice j, que l'on définit par:

$$i \in S_j \iff j \in P_i.$$

2.1.2 Le modèle étendu

Avec le modèle de programme décrit ci-après, nous cherchons à remédier à l'inconvénient majeur du modèle simple qui ne lui permet pas de prendre en compte sans modification des programmes où des variables intermédiaires seraient redéfinies. Cette situation se présente fréquemment. Comme exemple, donnons le cas du calcul du produit scalaire euclidien du vecteur x avec un vecteur $a \in \mathbb{R}^n$:

$$\left[\begin{array}{l} p := 0; \\ \textbf{for} \quad i := 1 \quad \textbf{to} \quad n \quad \textbf{do} \quad p := p + x_i * a_i. \end{array} \right.$$

Dans ce programme, la variable intermédiaire p est redéfinie n fois. Il est donc important d'avoir un modèle de programme acceptant naturellement de telles situations.

Dans le cadre du modèle étendu, nous noterons encore $x = (x_1, \ldots, x_n)$ le vecteur des n variables indépendantes et nous noterons $\bar{x} = (x_1, \ldots, x_n, \ldots, x_N)$ le vecteur formé de

toutes les variables intervenant dans le programme: variables indépendantes et intermédiaires. Dans ce modèle les fonctions intermédiaires φ_k peuvent dépendre de toutes les variables $x_1, \ldots, x_n, \ldots, x_N$. On numérote de 1 à K les fonctions intermédiaires et on note μ_k l'indice de la variable modifiée par l'instruction d'affectation d'ordre k. Avec ces conventions, le modèle étendu est schématisé par l'algorithme suivant:

$$\left[\begin{array}{l} \textbf{for} \quad k := 1 \quad \textbf{to} \quad K \quad \textbf{do} \quad x_{\mu_k} := \varphi_k(x_{P_k}); \\ f := x_N. \end{array}\right. \tag{2}$$

On a encore supposé que f était donnée par la variable x_N, ce qui n'est nullement restrictif puisque l'ordre dans lequel sont numérotées les variables intermédiaires est maintenant arbitraire. On notera $\bar{x}^0$ la valeur de $\bar{x}$ *avant* exécution du programme et $\bar{x}^k$ la valeur de $\bar{x}$ obtenue *après* avoir effectué k instructions d'affectation. Donc en fin de programme, $\bar{x}$ aura la valeur $\bar{x}^K$.

Nous allons faire sur le programme (2) l'hypothèse de consistance suivante: chaque instruction d'affectation n'utilise que des variables déjà définies par des instructions d'affectation précédentes. Afin d'être plus précis, désignons par ν_i l'indice de la *première* instruction qui donne une valeur à la variable x_i. Autrement dit, $k = \nu_i$ si et seulement si $\mu_k = i$ et $\mu_l \neq i$, pour $l = 1, \ldots, k-1$. Avec ces notations, l'*hypothèse de consistance* s'écrit:

$$\text{Pour} \quad n+1 \leq i \leq N \quad \text{et} \quad 1 \leq k \leq \nu_i, \quad \text{on a} \quad i \notin P_k. \tag{3}$$

Notons que dans ce modèle, nous ne faisons aucune hypothèse d'inviolabilité des variables indépendantes: on peut très bien avoir $1 \leq \mu_k \leq n$, pour certains k. En d'autres mots, les variables indépendantes peuvent être modifiées au cours du programme.

2.1.3 Choix des fonctions intermédiaires

D'un point de vue informatique, une fonction intermédiaire peut être beaucoup de choses: une fonction de calcul de base (`+`, `-`, `*`, `/`, `sin`, `cos`, `exp`, ...), une ligne de programme, un ensemble de lignes de programme, une sous-routine, etc. A la limite, ce peut être tout le programme calculant $f(x)$. Donc, s'il est clair qu'une fonction codable suivant le modèle simple ou étendu est aisément programmable sur ordinateur si les fonctions intermédiaires le sont, on a aussi la réciproque: toute fonction pouvant être calculée par un programme informatique est codable suivant le modèle simple ou étendu.

Afin de pouvoir implémenter aisément un différentiateur automatique, on est souvent amené à décomposer f en opérations élémentaires, de telle sorte que les fonctions intermédiaires φ_i soient les opérations-machine de base: addition, soustraction, multiplication, division, exponentielle, logarithme, fonctions trigonométriques, ... On notera $\mathcal{F}$ l'ensemble des fonctions intermédiaires utilisées. On distinguera la classe

$$\mathcal{F}_0 = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$$

des opérations arithmétiques élémentaires et la classe

$$\begin{aligned} \mathcal{F}_{\mathrm{F}} \quad = \quad \{&\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{sqrt}, \texttt{exp}, \texttt{log}, \texttt{log10}, \texttt{sin}, \texttt{cos}, \\ &\texttt{tan}, \texttt{asin}, \texttt{acos}, \texttt{atan}, \texttt{sinh}, \texttt{cosh}, \texttt{tanh}\} \end{aligned}$$

des fonctions accessibles en FORTRAN. Nous supposerons toujours que les fonctions de $\mathcal{F}$ sont régulières dans un voisinage du point où elles sont évaluées. Cette hypothèse n'est pas nécessairement vérifiée en pratique (on trouvera chez Kedem [38] et Fischer [15] quelques discussions sur ce sujet), mais elle permet de faciliter l'exposé des principes de la différentiation automatique.

Le plus souvent, les fonctions intermédiaires ne dépendront que d'une ou de deux variables. La plupart des implémentations actuelles de différentiateurs automatiques effectuent cette réduction en fonctions élémentaires. On en donnera les raisons ultérieurement.

Par exemple, la fonction

$$f(x) = \frac{x_1 e^{x_2} + x_3}{3 \sin x_4}$$

pourra être décomposée en

$$\left[\begin{array}{l} x_5 := \exp(x_2); \\ x_6 := x_1 * x_5; \\ x_7 := x_6 + x_3; \\ x_8 := \sin(x_4); \\ x_9 := 3 * x_8; \\ x_{10} := x_7 / x_9; \\ f := x_{10}. \end{array}\right.$$

On comprend que cette décomposition, qui permet de limiter le nombre de fonctions intermédiaires différentes à considérer, peut nécessiter l'introduction d'un grand nombre de variables intermédiaires.

2.2 La différentiation automatique en mode direct

Le mode direct de la différentiation automatique consiste à calculer la fonction et son gradient simultanément, en parallèle. Chaque fois qu'une variable intermédiaire est modifiée par l'instruction $x_{\mu_k} := \varphi_k(x_{P_k})$, son gradient ∇x_{μ_k} est calculé en utilisant la règle de dérivation des fonctions composées.

Cette approche remonte aux années 65-70: voir par exemple Wengert [71] pour une des premières implémentations *manuelles*. On trouvera dans le livre de Rall [57] et, par exemple, dans les articles de Kedem [38] et de Griewank [22] quelques notes sur les précurseurs.

En ce qui concerne la terminologie anglaise, on trouve *forward mode* (le plus courament), *forward accumulation* (Griewank [21]), ... On trouve aussi *bottom-up mode*, mais ce terme n'est pas à conseiller car il fait référence au graphe de calcul de le fonction, ce qui suppose un modèle simple de programme, et fait une convention sur le sens de construction de ce graphe (de bas en haut), qui n'est pas la même chez tous les auteurs.

2.2.1 L'algorithme DAD1

Les formules s'obtiennent aisément dans le cadre du modèle étendu de programme. L'idée est de "propager" le calcul du gradient des variables intermédiaires parallèlement à leur calcul.

Désignons par ∇x_i, $1 \leq i \leq N$, le gradient de la variable (indépendante ou intermédiaire) x_i par rapport aux variables indépendantes $x_1, \ldots, x_n$. Il s'agit donc de N vecteurs ayant chacun n composantes. On a bien sûr,

$$\nabla x_i = e_i, \quad \text{pour} \;\; 1 \leq i \leq n,$$

où $e_i := (0, \ldots, 0, 1, 0 \ldots, 0)^T$ est le i-ième vecteur de la base canonique de $\mathbb{R}^n$. Pour $1 \leq k \leq K$, la définition de x_{μ_k} et la règle de dérivation des fonctions composées donnent:

$$\nabla x_{\mu_k} = \sum_{j \in P_k} \frac{\partial \varphi_k}{\partial x_j} \nabla x_j, \quad 1 \leq k \leq K. \tag{4}$$

Le mode direct consiste à faire ce type de calcul pour chaque instruction du code original. Par convention, la formule (4) s'écrit $\nabla x_{\mu_k} = 0$ si $P_k = \emptyset$. Dès lors, $\nabla x_i = 0$ chaque fois qu'une variable intermédiaire x_i est initialisée.

Remarquons que si le code (2) est consistant au sens de (3), le membre de droite dans (4) est connu au moment du calcul de x_{μ_k}. En effet , d'après cette hypothèse de consistance, les variables x_j, $j \in P_k$, ont été calculées avant l'étape k. Il en sera donc de même de leur gradient.

Si on note $\varphi_{k,j} \equiv \partial \varphi_k / \partial x_j$, l'algorithme de calcul du gradient ∇f s'écrit:

$$\left[\begin{array}{l} \textbf{for} \;\; i := 1 \;\; \textbf{to} \;\; n \;\; \textbf{do} \;\; \nabla x_i := e_i; \\ \textbf{for} \;\; k := 1 \;\; \textbf{to} \;\; K \;\; \textbf{do} \;\; \{ \\ \quad \nabla x_{\mu_k} := \sum_{j \in P_k} \varphi_{k,j} \nabla x_j; \\ \quad x_{\mu_k} := \varphi_k(x_{P_k}); \\ \} \\ \nabla f := \nabla x_N; \\ f := x_N. \end{array}\right.$$

On voit qu'il suffit de compléter le programme original par quelques instructions supplémentaires sans modifier le sens de l'exécution. On dit que le calcul du gradient est propagé de *façon progressive*.

Notons qu'il est important de calculer ∇x_{μ_k} avant de calculer x_{μ_k}, parce que φ_k peut dépendre de x_{μ_k} et que ses dérivées partielles doivent être évaluées en $\bar{x}^{k-1}$ et non pas en $\bar{x}^k$ (voir Section 2.1.2 pour les notations).

On voit à présent l'intérêt d'avoir des fonctions intermédiaires φ_k connues *a priori*: on connaît alors également *a priori* les formules donnant les $\varphi_{k,j}$, sans avoir à faire une analyse syntaxique des fonctions φ_k.

2.2.2 L'algorithme DAD2

On peut également calculer les dérivées partielles $\partial f / \partial x_i$ successivement, pour $i = 1, \ldots, n$: le temps d'exécution est plus important (il faut recalculer chaque fois $\varphi_k(x_{P_k})$), mais l'espace mémoire nécessaire est nettement moindre. Dans le programme suivant, on a noté u_k la valeur (instantanée) de la dérivée partielle $\partial x_k / \partial x_i$, $1 \leq k \leq N$, $1 \leq i \leq n$:

$$
\left[\begin{array}{l}
\textbf{for}\ \ i := 1\ \ \textbf{to}\ \ n\ \ \textbf{do}\ \ \{ \\
\quad \textbf{for}\ \ l := 1\ \ \textbf{to}\ \ n\ \ \textbf{do}\ \ u_l := 0; \\
\quad u_i := 1; \\
\quad \textbf{for}\ \ k := 1\ \ \textbf{to}\ \ K\ \ \textbf{do}\ \ \{ \\
\qquad u_{\mu_k} := \sum_{j \in P_k} \varphi_{k,j}\, u_j; \\
\qquad x_{\mu_k} := \varphi_k(x_{P_k}); \\
\quad \} \\
\quad \frac{\partial f}{\partial x_i} := u_N; \\
\quad f := x_N; \\
\}.
\end{array}\right. \tag{5}
$$

Cette manière de procéder est dans son esprit assez proche du calcul du gradient par différences divisées: pour chaque direction de base e_i de $\mathbb{R}^n$, on calcule f et $\partial f/\partial x_i$. C'est un peu plus coûteux que le calcul approché (qui demande $n+1$ fois le calcul de la fonction), mais c'est nettement plus précis.

L'algorithme précédent s'adapte particulièrement bien au calcul de la dérivée d'une fonction dans une direction $u = (u_1, \ldots, u_n)$ donnée: $f'(x) \cdot u = \nabla f(x)^T u$. Les composantes $u_1, \ldots, u_n$ étant données, il suffit de calculer $u_{\mu_k} \equiv x'_{\mu_k}(x) \cdot u$, $1 \leq k \leq K$, par la boucle la plus interne de l'algorithme précédent. Ceci donne:

$$
\left[\begin{array}{l}
\textbf{for}\ \ k := 1\ \ \textbf{to}\ \ K\ \ \textbf{do}\ \ \{ \\
\quad u_{\mu_k} := \sum_{j \in P_k} \varphi_{k,j}\, u_j; \\
\quad x_{\mu_k} := \varphi_k(x_{P_k}); \\
\} \\
f'(x) \cdot u := u_N; \\
f := x_N; \\
\}.
\end{array}\right. \tag{6}
$$

2.2.3 L'algorithme DAD3

L'algorithme DAD2 demande que l'on recalcule la valeur des variables intermédiaires, pour l'évaluation de chaque dérivée partielle, ce qui est coûteux en temps de calcul. Ceci peut être évité si le programme de calcul de la fonction f obéit au modèle simple (1) et si on mémorise son graphe de calcul (ce qui est coûteux en espace mémoire !).

Représentons le graphe de calcul de f, en pondérant chaque arc par les valeurs $\varphi_{k,j} \equiv \partial\varphi_k/\partial x_j$ (Figure 2). Dans la boucle d'indice i de l'algorithme DAD2 (5), on associe à chaque noeud k, la valeur de la variable x_k, ainsi que la valeur u_k de sa dérivée partielle par rapport à x_i. Au départ, les valeurs de x_k, $1 \leq k \leq n$, sont données et les valeurs de u_k sont initialisées à δ_{ki} (symbole de Kronecker: $\delta_{ki} = 1$ si $k = i$ et $\delta_{ki} = 0$ si $k \neq i$). Ensuite, les noeuds du graphe sont parcourus séquentiellement, pour $k = n+1, \ldots, N$, et la valeur des variables $x_k := \varphi_k(x_{P_k})$ et de leur dérivée partielle $u_k := \sum_{j \in P_k} \varphi_{k,j}\, u_j$, ainsi que les pondérations $\varphi_{k,j}$ ($j \in P_k$) sont calculées.

Supposons à présent que, lors de l'exécution du programme (1), on construise son graphe de calcul avec ses valeurs de x_k et de $\varphi_{k,j}$. Alors, le temps de calcul de la i-ième dérivée partielle sera moindre si on ne parcourt que la partie du graphe dépendant de la variable x_i. Pour cela, au lieu de parcourir le graphe en regardant pour chaque noeud k quelles sont les

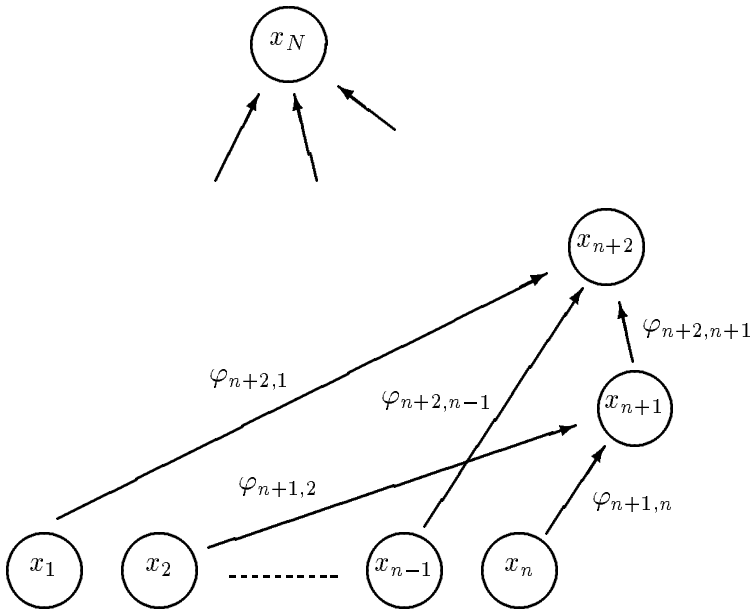

Figure 2: Graphe de calcul avec poids.

variables x_j dont sa valeur dépend ($j \in P_k$), il suffit de regarder quelles sont les variables x_j qu'il influence ($j \in S_k$). On enregistre alors, pour $j \in S_k$, la contribution de u_k à la dérivée partielle u_j de x_j: $u_j := u_j + \varphi_{j,k} u_k$. Comme le graphe est parcouru séquentiellement, pour $k = 1, \ldots, N$, la valeur de u_k dans la formule précédente aura sa valeur définitive au moment de son utilisation puisque seuls des noeuds d'indice inférieur à k peuvent influencer cette valeur. On obtient alors l'algorithme suivant pour le calcul de $\partial f / \partial x_i$:

```
for i := 1 to n do {
    u_i := 1;
    for l := i + 1 to N do u_l := 0;
    for k := i to N - 1 do
        if u_k ≠ 0 then for j ∈ S_k do u_j := u_j + φ_{k,j} u_k;
    }
    ∂f/∂x_i := u_N;
}.
```

C'est le test "**if** $u_k \neq 0$" qui permet de ne parcourir qu'une partie du graphe ou, plus exactement, qui permet d'éviter de faire des opérations inutiles ($|S_k|$ additions et $|S_k|$ multiplications, où $|S_k|$ est le nombre d'éléments de S_k) lorsque la contribution du noeud k est nulle.

Il y a deux différences fondamentales entre les algorithmes DAD1 (ou DAD2) et DAD3:

- Dans le premier (plus particulièrement DAD1), la fonction f et ses dérivées partielles sont calculées en parallèle, tandis que le second algorithme se déroule en deux phases: calcul de f avec construction du graphe, suivi du calcul des dérivées.
- Le premier algorithme (plus particulièrement DAD2) est relativement économe en place

mémoire, alors que le second doit enregistrer toute la *trace* de la première phase dans une base de données (le graphe) qui peut être très encombrante en mémoire.

Nous retrouverons les caractéristiques de DAD3 dans les algorithmes utilisant le mode inverse de différentiation automatique.

2.2.4 Temps d'exécution et encombrement mémoire

On note respectivement $T(+)$, $T(*)$, $T(\varphi_i)$ et $T(\varphi_{i,j})$ les temps d'exécution d'une addition, d'une multiplication et le temps nécessaire au calcul de φ_i et de $\partial\varphi_i/\partial x_j$. On note $|P_i|$ le nombre d'éléments de P_i. On a alors le résultat suivant.

Proposition 2.1 *Supposons qu'il existe une constante positive γ telle que $T(\varphi_k) \leq \gamma|P_k|$, $\forall\varphi_k \in \mathcal{F}$. Alors le temps $T_{\mathrm{DAD1}}(f,\nabla f)$ nécessaire au calcul de f et ∇f par le programme DAD1 vérifie l'estimation suivante:*

$$\frac{T_{\mathrm{DAD1}}(f,\nabla f)}{T_{(2)}(f)} \geq 1 + \left(\frac{T(*)}{\gamma}\right) n, \tag{7}$$

où $T_{(2)}(f)$ est le temps de calcul nécessaire au calcul de f par l'algorithme (2).

Preuve. En utilisant la constante γ, on obtient une borne supérieure pour $T_{(2)}(f)$:

$$T_{(2)}(f) = \sum_{k=1}^{K} T(\varphi_k) \leq \gamma \sum_{k=1}^{K} |P_k|.$$

On peut alors minorer le temps $T_{\mathrm{DAD1}}(f,\nabla f)$ comme suit:

$$\begin{aligned} T_{\mathrm{DAD1}}(f,\nabla f) &= \sum_{k=1}^{K} \left(T(\varphi_k) + n(|P_k|-1)T(+) + n|P_k|T(*) + \sum_{j\in P_k} T(\varphi_{k,j}) \right) \\ &\geq T_{(2)}(f) + n\left(\sum_{k=1}^{K} |P_k|\right) T(*) \\ &\geq T_{(2)}(f)\left(1 + \frac{nT(*)}{\gamma}\right), \end{aligned}$$

ce qui donne (7). □

La rapport entre les deux temps d'exécution est donc au moins proportionnel à n. On voit que ceci est principalement dû aux n multiplications nécessaire au calcul de $\varphi_{k,j}\nabla x_j$ dans l'algorithme DAD1. Remarquons que ce quotient en au moins $\mathcal{O}(n)$ est encore plus visible pour l'algorithme DAD2, puisque l'on a:

$$T_{\mathrm{DAD2}}(f,\nabla f) = n\left(\sum_{i=n+1}^{N} \left[T(\varphi_i) + (|P_i|-1)T(+) + |P_i|T(*) + \sum_{j\in P_i} T(\varphi_{i,j}) \right]\right) \geq nT_{(2)}(f).$$

Cette estimation du temps de calcul par le mode direct n'est valable que pour l'algorithme DAD1, qui est un algorithme général, non à même de mettre à profit le caractère creux

éventuel de la fonction calculée. Si, dans (4), beaucoup de composantes de ∇x_j sont nulles, il est inutile de faire n multiplications. Dès lors, l'estimation (7) peut ne pas être valable pour des implémentations *manuelles* du mode direct, adaptées à une fonction particulière.

Il est également intéressant d'avoir une estimation du coût relatif du calcul d'une dérivée directionnelle par l'algorithme (6). C'est l'objet de la proposition suivante.

Proposition 2.2 *Le temps $T_{\text{DAD2-(6)}}(f, f'(x)\cdot u)$ nécessaire au calcul de f et de sa dérivée directionnelle $f'(x)\cdot u$ dans la direction u au moyen de l'algorithme DAD2-(6) vérifie l'estimation suivante:*

$$\frac{T_{\text{DAD2-(6)}}(f, f'(x)\cdot u)}{T_{(2)}(f)} \leq C_{\mathcal{F}},$$

où $T_{(2)}(f)$ est le temps nécessaire au calcul de f par l'algorithme (2) et $C_{\mathcal{F}}$ est une constante ne dépendant que des fonctions de la bibliothèque $\mathcal{F}$.

Preuve. D'après l'algorithme (6), on a

$$T_{\text{DAD2-(6)}}(f, f'(x)\cdot u) = \sum_{k=1}^{K} T_{\text{DAD2-(6)}}(\varphi_k, \varphi_k'(x_{P_k})\cdot u).$$

Ensuite, en utilisant l'inégalité de Hölder, on obtient

$$\begin{aligned} T_{\text{DAD2-(6)}}(f, f'(x)\cdot u) &\leq \max_{1\leq k\leq K}\left(\frac{T_{\text{DAD2-(6)}}(\varphi_k, \varphi_k'(x_{P_k})\cdot u)}{T(\varphi_k)}\right)\sum_{k=1}^{K} T(\varphi_k) \\ &\leq \max_{\varphi\in\mathcal{F}}\left(\frac{T_{\text{DAD2-(6)}}(\varphi, \varphi'(x_P)\cdot u)}{T(\varphi)}\right) T_{(2)}(f) \\ &= C_{\mathcal{F}} T_{(2)}(f), \end{aligned}$$

où la constante

$$C_{\mathcal{F}} = \max_{\varphi\in\mathcal{F}}\left(\frac{T_{\text{DAD2-(6)}}(\varphi, \varphi'(x_P)\cdot u)}{T(\varphi)}\right)$$

ne dépend que de la classe $\mathcal{F}$ des fonctions intermédiaires. □

Il peut être utile d'avoir une estimation plus précise de la borne $C_{\mathcal{F}}$, pour certaines familles $\mathcal{F}$ de fonctions. Par exemple, si on suppose que toutes les fonctions intermédiaires φ_k sont dans la classe $\mathcal{F}_0 = \{+, -, *, /\}$ des opérations arithmétiques élémentaires, on montre facilement que l'on a

$$C_{\mathcal{F}_0} \leq \max\left(2, 3 + \frac{T(+)}{T(*)}, 2 + \frac{T(-)+T(*)}{T(/)}\right),$$

où le premier argument du max correspond aux opérations + ou -, le second à * et le troisième à /. Par exemple, pour la division $x_k := x_i/x_j$, la dérivée directionnelle u_k de x_k peut s'obtenir par $u_k := (u_i - x_k u_j)/x_j$ et donc sa contribution à $C_{\mathcal{F}_0}$ s'écrit

$$\frac{4T(\boxplus) + 2T(\boxminus) + T(-) + T(*) + 2T(/)}{2T(\boxplus) + T(\boxminus) + T(/)} \leq 2 + \frac{T(-)+T(*)}{T(/)},$$

où $T(\boxed{\rightarrow})$ et $T(\boxed{\leftarrow})$ représentent respectivement les temps qu'il faut pour aller chercher et pour sauvegarder un nombre flottant en mémoire. Dès lors, comme il est raisonnable de supposer que $T(\texttt{+}) \leq T(\texttt{*})$ et que $T(\texttt{-})$ et $T(\texttt{*})$ sont tous deux majorés par $T(\texttt{/})$, on a certainement l'estimation

$$C_{\mathcal{F}_0} \leq 4.$$

Si on considère la classe $\mathcal{F}_{\mathrm{F}}$ du FORTRAN, on peut montrer que, sous des hypothèses raisonnables telles que $T(\varphi_0) \leq T(\varphi_1)$, $\forall \varphi_0 \in \mathcal{F}_0$ et $\forall \varphi_1 \in \mathcal{F}_{\mathrm{F}} \backslash \mathcal{F}_0$, $T(\texttt{*}) + T(\texttt{cos}) \leq 2T(\texttt{sin})$, $T(\texttt{/}) + T(\texttt{cos}) \leq 2T(\texttt{asin})$, ..., on garde la majoration

$$C_{\mathcal{F}_{\mathrm{F}}} \leq 4.$$

Quant à l'espace mémoire *supplémentaire* requis, il est en $\mathcal{O}(nN)$ pour l'algorithme DAD1 et en $\mathcal{O}(N)$ pour l'algorithme DAD2. On notera que dans ces deux algorithmes, il n'est pas nécessaire de mémoriser les quantités $\varphi_{i,j} \equiv \partial \varphi_i / \partial x_j$, puisque celles-ci sont utilisées directement après leur calcul. Une analyse fine du programme pourrait éventuellement permettre d'économiser de la place en mémoire en réutilisant l'espace alloué aux dérivées (ou aux gradients) des variables mortes (celles qui n'interviennent plus) pour les dérivées des variables naissantes. Enfin, pour l'algorithme DAD3, il est également nécessaire de mémoriser le graphe de calcul et l'espace mémoire supplémentaire sera en $\mathcal{O}(N) + \mathcal{O}(K)$.

2.3 La différentiation automatique en mode inverse

Nous présentons dans cette section un autre jeu de règles permettant de construire, de manière automatique, un programme calculant les dérivées partielles de la fonction définie par l'algorithme (1) ou (2). Cette méthode, qui s'apparente au calcul du gradient par la méthode de l'état adjoint dans les problèmes de contrôle optimal, a deux intérêts majeurs. D'une part, elle donne des informations sur la variation de la fonction calculée lorsque les variables intermédiaires varient. Ceci sera utile pour l'étude de la propagation des erreurs d'arrondi (Section 3). D'autre part, l'algorithme qui s'en inspire a un quotient $T(f, \nabla f)/T(f)$ majoré par une constante indépendante de n.

Ce mode de dérivation remonte au moins à Linnainmaa (1976) qui s'intéressait à l'analyse des erreurs d'arrondi. Dans le cadre de la différentiation automatique de programmes, Speelpenning (1980) semble être le premier à avoir introduit ce mode de dérivation. La méthode a ensuite été redécouverte par de nombreux auteurs dont Iri (1984), Kim, Nesterov et Cherkasskiĭ (1984), Sawyer (1984), ...

En ce qui concerne la terminologie anglaise, on trouve *fast automatic differentiation*, *backward accumulation*, ..., mais l'appellation qui semble se généraliser est "*backward mode*". Pour les mêmes raisons que précédemment (Section 2.2), le terme *top-down mode* ne nous semble pas approprié.

Ce mode de dérivation étant moins intuitif que le mode direct, nous en donnerons plusieurs approches. Viendront en premier lieu celles qui ne peuvent être utilisées que pour des programmes basés sur le modèle simple. Cela permettra d'introduire l'algorithme DAI1. Ensuite, nous donnerons d'autres approches, ou des extensions des précédentes, qui peuvent prendre en compte des programmes basés sur le modèle étendu. Cela nous donnera l'algorithme DAI2, qui diffère quelque peu de l'algorithme DAI1.

2.3.1 Approche par le graphe de calcul

Cette approche est utilisée par Sawyer (1984), Iri (1984) et Iri et Kubota (1987) pour la dérivation des formules. C'est aussi sur cette approche que sont construits les différentiateurs automatiques PADRE2 [32] et ADOL-C [26].

Reprenons le graphe de calcul (figure 2) de f. D'après la formule de dérivation des fonctions composées, la dérivée partielle $\partial f/\partial x_k$ peut être obtenue en faisant la somme sur tous les chemins menant de x_k à x_N, des produits des quantités $\varphi_{i,j}$ associées aux arcs rencontrés sur chaque chemin. Pour voir cela, il suffit d'appliquer répétitivement la formule (4), avec f au lieu de x_{μ_k} et en n'en considérant que la composante k. On a:

$$\frac{\partial f}{\partial x_k} = \sum_{\substack{\text{chemin } \mathcal{C} \\ \text{de } x_k \text{ à } x_N}} \prod_{\text{arc } (i,j)\in\mathcal{C}} \varphi_{i,j} = \sum_{\substack{\text{chemin } \mathcal{C} \\ \text{de } x_k \text{ à } x_N}} \varphi_{N,\cdot} \cdots \varphi_{\cdot,k}. \tag{8}$$

Le mode inverse, introduit à partir du graphe de calcul, est basé sur l'observation suivante. Lorsque l'on applique la formule précédente pour $k = 1, \ldots, n$, il y a des produits de $\varphi_{i,j}$ qui sont calculés plusieurs fois. Cela apparaît clairement dans le graphe particulier de la figure 3 ($N = n + 3$) dans lequel la quantité $(\varphi_{N,n+2}\, \varphi_{n+2,n+1})$ apparaît dans le calcul de chaque

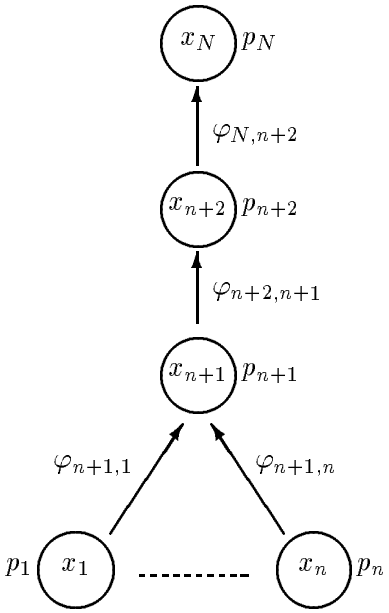

Figure 3: Un graphe de calcul particulier.

dérivées partielles de f:

$$\frac{\partial f}{\partial x_k} = (\varphi_{N,n+2}\, \varphi_{n+2,n+1})\, \varphi_{n+1,k}.$$

Dans ce cas particulier, on évitera le calcul répétitif de $(\varphi_{N,n+2}\, \varphi_{n+2,n+1})$ en accumulant dans des variables p_i associées aux noeuds x_i, les produits

$$p_N = 1$$

$$
\begin{aligned}
p_{n+2} &= \varphi_{N,n+2}\, p_N \\
p_{n+1} &= \varphi_{n+2,n+1}\, p_{n+2} \\
\frac{\partial f}{\partial x_k} &= \varphi_{n+1,k}\, p_{n+1}, 1 \leq k \leq n.
\end{aligned}
$$

Cette manière de faire correspond à un parcourt du graphe de haut en bas, à partir du sommet x_N. Bien entendu, on ne gagnera pas à tous les coups, en particulier si le graphe de calcul est articulé comme à la figure 4. Cependant, dans le cas fréquent où $|P_i| \ll n$, la

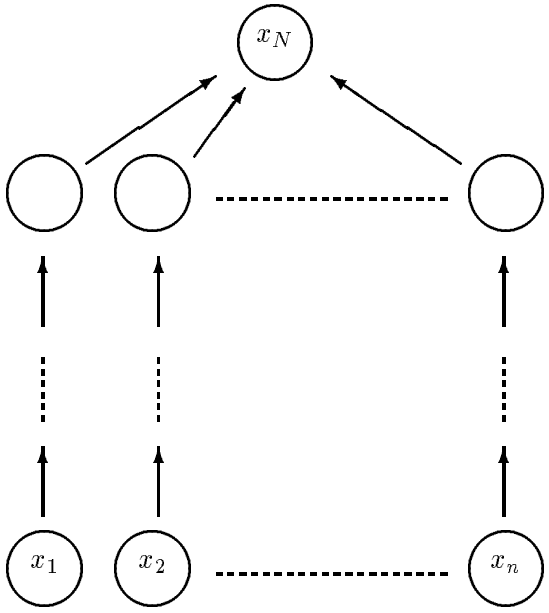

Figure 4: Autre graphe de calcul particulier.

procédure précédente sera généralement avantageuse.

Pour généraliser cette procédure à un graphe de calcul quelconque, on associe à chaque noeud x_k $(1 \leq k \leq N)$, la variable p_k définie par

$$
p_k = \sum_{\substack{\text{chemin } \mathcal{C} \\ \text{de } x_k \text{ à } x_N}} \prod_{\text{arc } (i,j) \in \mathcal{C}} \varphi_{i,j}.
$$

Cette formule est analogue à celle donnant $\partial f/\partial x_k$. On a d'ailleurs $\partial f/\partial x_k = p_k$, pour $1 \leq k \leq n$. On voit que p_k représente la variation de f lorsque la variable intermédiaire x_k varie, les autres variables intermédiaires x_i, $k < i \leq N$, variant progressivement via les fonctions intermédiaires φ_i. On en déduit que l'on peut déterminer les variables p_k de *manière régressive*, pour $k = N, \ldots, 1$. On a en effet

$$
p_N = 1.
$$

Ensuite, si les variables p_i ont été initialisés à 0, il suffit de parcourir le graphe de haut en bas et à chaque noeud k, de prendre en compte la contribution de p_k à p_j, pour $j \in P_k$. Elle

s'écrit

$$p_j = p_j + \frac{\partial \varphi_k}{\partial x_j} p_k, \quad \forall j \in P_k.$$

Ces considérations conduisent à l'algorithme suivant:

$$\left[\begin{array}{l} \textbf{for} \quad i := 1 \quad \textbf{to} \quad N-1 \quad \textbf{do} \quad p_i := 0; \\ p_N := 1; \\ \textbf{for} \quad k := N \quad \textbf{down to} \quad n+1 \quad \textbf{do} \\ \quad \textbf{for} \quad j \in P_k \quad \textbf{do} \quad p_j := p_j + \varphi_{k,j}\, p_k. \end{array} \right.$$

En fin de calcul, la dérivée partielle $\partial f / \partial x_i$, $1 \leq i \leq n$, se récupère dans p_i.

2.3.2 Approche par substitutions régressives I

L'approche que nous développons ici est utilisée par Speelpenning (1980, Section 4.6) et Kim, Nesterov et Cherkasskiĭ (1984). C'est aussi essentiellement l'idée utilisée par Morgenstern (1985) pour obtenir une borne supérieure sur le coût du calcul d'un gradient.

Soit i un indice fixé entre n et $N-1$. On s'intéresse à la manière dont la variable x_N (c'est-à-dire la fonction calculée) varie lorsque x_i varie, les variables $x_1, \ldots, x_{i-1}$ étant fixées à des valeurs données et la variation des variables $x_{i+1}, \ldots, x_N$ se faisant par l'intermédiaire des fonctions $\varphi_{i+1}, \ldots, \varphi_N$. On est donc amené à considérer la fonction $\psi_i(x_1, \ldots, x_i)$ définie en utilisant les $N-i$ dernières instructions du programme (1):

$$\left[\begin{array}{l} \textbf{for} \quad k := i+1 \quad \textbf{to} \quad N \quad \textbf{do} \quad x_k := \varphi_k(x_1, \ldots, x_{k-1}); \\ \psi_i := x_N. \end{array} \right.$$

On définit également

$$\psi_N(x_1, \ldots, x_N) = x_N. \tag{9}$$

Alors, pour $k = N-1, N-2, \ldots, n$, on a les formules de récurrence *régressives* suivantes:

$$\psi_k(x_1, \ldots, x_k) = \psi_{k+1}(x_1, \ldots, x_k, \varphi_{k+1}(x_1, \ldots, x_k)). \tag{10}$$

Par définition de f, ces substitutions conduisent à

$$\psi_n(x_1, \ldots, x_n) = f(x_1, \ldots, x_n). \tag{11}$$

On note

$$p_i(x_1, \ldots, x_i) \equiv \frac{\partial \psi_i}{\partial x_i}(x_1, \ldots, x_i), \tag{12}$$

la dérivée cherchée. Lorsque la valeur des variables $x_1, \ldots, x_n$ est donnée et que les variables intermédiaires $x_{n+1}, \ldots, x_N$ sont calculées par l'algorithme (1), on note $p_i \equiv p_i(x_1, \ldots, x_i)$ sans mentionner explicitement la dépendance de p_i en $x_1, \ldots, x_i$.

Les formules (9), (10) et (11) permettent de calculer de façon régressive $p_N, p_{N-1}, \ldots, p_{n+1}$ et ensuite les dérivées partielles de f par rapport à $x_n, x_{n-1}, \ldots, x_1$. En effet, à partir de (9), on obtient

$$p_N = 1. \tag{13}$$

Ensuite, si $1 \leq i \leq N-1$ et $\max(i,n) \leq k \leq N-1$, on a en dérivant (10):

$$\frac{\partial \psi_k}{\partial x_i}(x_1,\ldots,x_k) = \frac{\partial \psi_{k+1}}{\partial x_i}(x_1,\ldots,x_{k+1}) + p_{k+1}(x_1,\ldots,x_{k+1})\,\frac{\partial \varphi_{k+1}}{\partial x_i}(x_1,\ldots,x_k), \qquad (14)$$

où $x_{k+1} = \varphi_{k+1}(x_1,\ldots,x_k)$. Supposons d'abord que $n+1 \leq i \leq N-1$. Alors, en sommant les équations précédentes pour $k = i,\ldots,N-1$ et en tenant compte de ce que $\partial\psi_N/\partial x_i = 0$, on trouve les formules de p_i:

$$p_i = \sum_{k=i+1}^{N} \frac{\partial \varphi_k}{\partial x_i}(x_1,\ldots,x_{k-1})\,p_k, \quad n+1 \leq i \leq N-1. \qquad (15)$$

Supposons à présent que $1 \leq i \leq n$. Alors, en sommant les équations (14) pour $k = n,\ldots,N-1$, on obtient

$$\frac{\partial \psi_n}{\partial x_i} = \sum_{k=n}^{N-1} \frac{\partial \varphi_{k+1}}{\partial x_i}(x_1,\ldots,x_k)\,p_{k+1}.$$

Grâce à (11), on trouve

$$\frac{\partial f}{\partial x_i} = \sum_{k=n+1}^{N} \frac{\partial \varphi_k}{\partial x_i}(x_1,\ldots,x_{k-1})\,p_k, \quad 1 \leq i \leq n, \qquad (16)$$

qui permet le calcul de ∇f à partir des p_k.

Avec (13), (15) et (16) on retrouve l'algorithme précédent.

2.3.3 Approche par dualité I

Cette approche permet, en faisant intervenir la notion de lagrangien, d'interpréter les variables p_i introduites précédemment comme des multiplicateurs de Lagrange. Elle permet également d'établir un lien entre le mode inverse et le calcul du gradient par la méthode de l'état adjoint dans les problèmes de contrôle optimal.

L'idée est la suivante. Au lieu de considérer la fonction f donnée par l'algorithme (1), fonction des n variables $x = (x_1,\ldots,x_n)$, on considère une fonction $\bar{f}$ dépendant des N variables $\bar{x} = (x_1,\ldots,x_n,\ldots,x_N)$, définie par $\bar{f}(\bar{x}) = x_N$, les variables $(x_1,\ldots,x_N)$ étant liées par les relations suivantes:

$$c_i(\bar{x}) \equiv x_i - \varphi_i(x_{P_i}) = 0, \text{ pour } i = n+1,\ldots,N. \qquad (17)$$

Il faut vérifier ici que le fait de remplacer les *instructions* d'affectation $x_i := \varphi_i(x_{P_i})$ par les *équations* (17) est licite. C'est-à-dire qu'étant données des valeurs aux variables indépendantes $x_1,\ldots,x_n$, le système d'équations (17) admet une solution unique $(x_{n+1},\ldots,x_N)$ telle que la valeur de x_N soit identique à celle obtenue par l'algorithme (1). L'existence d'une solution de (17) vérifiant cette condition est claire: il suffit de résoudre les équations pas à pas pour $i = n+1,\ldots,N$ comme le fait l'algorithme (1). Pour la même raison, il est également clair qu'il ne peut y avoir d'autres solutions.

Dans cette formulation, il s'agit de calculer le gradient d'une fonction $\bar{f}$ de N variables, celles-ci étant assujetties aux contraintes (17). Cette situation se présente par exemple dans

les problèmes de contrôle optimal. De ce point de vue, les variables indépendantes $x_1, \ldots, x_n$ jouent le rôle de *variables de contrôle* et les variables dépendantes $x_{n+1}, \ldots, x_N$ jouent le rôle de *variables d'état*. Pour calculer le gradient de $\bar{f}$ par rapport aux variables de contrôle, il est commode d'utiliser la méthode de l'état adjoint. On associe un *multiplicateur de Lagrange* p_i $(i = n+1, \ldots, N)$ à chaque contrainte et on introduit le *lagrangien* ℓ, défini par

$$\ell(\bar{x}, p) \equiv \bar{f}(\bar{x}) - \sum_{k=n+1}^{N} c_k(\bar{x})\, p_k = x_N - \sum_{k=n+1}^{N} (x_k - \varphi_k(x_{P_k}))\, p_k,$$

où le vecteur $p \equiv (p_{n+1}, \ldots, p_N)$ s'appelle également l'*état adjoint*.

En un point $\bar{x}$ donné, on calcule l'état adjoint p au moyen des *équations adjointes*:

$$\frac{\partial \ell}{\partial x_i}(\bar{x}, p) = 0, \text{ pour } i = n+1, \ldots, N.$$

On trouve pour $i = n+1, \ldots, N$:

$$\sum_{k=n+1}^{N} (\delta_{ik} - \frac{\partial \varphi_k}{\partial x_i})\, p_k = \delta_{iN},$$

où δ est le symbole de Kronecker: $\delta_{ij} = 1$ si $i = j$ et $\delta_{ij} = 0$ si $i \neq j$. Comme $\partial \varphi_k / \partial x_i = 0$ pour $i \geq k$, ces équations forment un système linéaire d'ordre $N - n$ en p dont la matrice est triangulaire supérieure avec une diagonale formée de 1. Ce système est donc inversible (ce qui assure du même coup la validité de la méthode de l'état adjoint) et peut se résoudre *de façon régressive* comme suit:

$$\begin{cases} p_N = 1, \\ p_i = \displaystyle\sum_{k=i+1}^{N} \frac{\partial \varphi_k}{\partial x_i}\, p_k, \text{ pour } i = N-1, \ldots, n+1. \end{cases} \tag{18}$$

On retrouve les formules (13) et (15).

Les dérivées partielles par rapport à $x_1, \ldots, x_n$ sont les mêmes pour f ou $\bar{f}$ et s'obtiennent par les formules

$$\frac{\partial f}{\partial x_i}(x) = \frac{\partial \ell}{\partial x_i}(\bar{x}, p), \text{ pour } i = 1, \ldots, n.$$

On trouve

$$\frac{\partial f}{\partial x_i} = \sum_{k=n+1}^{N} \frac{\partial \varphi_k}{\partial x_i}\, p_k, \text{ pour } i = 1, \ldots, n, \tag{19}$$

formules identiques à (16). C'est la même formule que celle donnant les multiplicateurs puisque $\partial \varphi_k / \partial x_i = 0$ lorsque $1 \leq i < k \leq n$.

2.3.4 L'algorithme DAI1

Le calcul de la fonction et de ses dérivées partielles se fait en deux étapes. Dans la première étape (*phase progressive*), on calcule par l'algorithme (1) la valeur de f via les variables intermédiaires $x_{n+1}, \ldots, x_N$. Il faut aussi calculer et *mémoriser* les quantités $\varphi_{i,j} \equiv \partial \varphi_i / \partial x_j$ qui

serviront dans la seconde phase. Dans la seconde étape (*phase régressive*), on calcule p et ∇f par (18)-(19).

Phase progressive

$$\left[\begin{array}{l} \textbf{for}\ \ i := n+1\ \ \textbf{to}\ \ N\ \ \textbf{do}\ \ \{ \\ \quad \varphi_{i,j} := \frac{\partial \varphi_i}{\partial x_j},\ \ \forall j \in P_i; \\ \quad x_i := \varphi_i(x_{P_i}); \\ \} \\ f := x_N. \end{array}\right. \tag{20}$$

Phase régressive

$$\left[\begin{array}{l} \textbf{for}\ \ i := 1\ \ \textbf{to}\ \ N-1\ \ \textbf{do}\ \ p_i := 0; \\ p_N := 1; \\ \textbf{for}\ \ k := N\ \ \textbf{down to}\ \ n+1\ \ \textbf{do} \\ \quad \textbf{for}\ \ j \in P_k\ \ \textbf{do}\ \ p_j := p_j + \varphi_{k,j}\, p_k; \\ \textbf{for}\ \ i := 1\ \ \textbf{to}\ \ n\ \ \textbf{do}\ \ \frac{\partial f}{\partial x_i} := p_i. \end{array}\right. \tag{21}$$

Dans la seconde phase, on a profité du fait que (18) et (19) ont la même structure et on a modifié l'ordre de sommation. Au lieu d'appliquer directement (18)-(19), on considère successivement la contribution de chaque fonction intermédiaire φ_k, pour $k = N, \ldots, n+1$, à tous les p_i. Comme φ_k n'influence p_i que pour les indices $i < k$, l'utilisation de p_k dans la formule est licite puisqu'il a déjà sa valeur correcte au moment de son utilisation. Cette manière de procéder accélère le calcul puisque P_k contient en général peu d'éléments. On retrouve ainsi l'algorithme de la section 2.3.1.

2.3.5 Approche de Speelpenning

Plaçons-nous à présent dans le cadre du modèle étendu (2) de programmes. La première approche que nous considérerons est due à Speelpenning (1980). Elle est particulièrement simple. L'idée est de voir une instruction d'affectation, $x_{\mu_k} := \varphi_k(x_{P_k})$, comme une transformation de toutes les variables du code, $\Phi_k : \mathbb{R}^N \to \mathbb{R}^N$, laissant inchangées les variables autres que x_{μ_k}. L'application Φ_k est définie par

$$(\Phi_k(\bar{x}))_i = \left\{ \begin{array}{ll} \bar{x}_i & \text{si} \quad i \neq \mu_k \\ \varphi_k(\bar{x}_{P_k}) & \text{si} \quad i = \mu_k. \end{array}\right.$$

Avec les notations de la section 2.1.2, on a $\bar{x}^k = \Phi_k(\bar{x}^{k-1})$, pour $k = 1, \ldots, K$. Dès lors, la valeur $\bar{x}^K$ des variables calculées par le programme est obtenue en appliquant à $\bar{x}^0$ la composée des K applications Φ_k, $1 \leq k \leq K$:

$$\bar{x}^K = (\Phi_K \circ \cdots \circ \Phi_1)(\bar{x}^0).$$

Alors, le gradient de f s'obtient en sélectionnant les n premières colonnes de la dernière ligne de la jacobienne de la transformation totale $\Phi_K \circ \cdots \circ \Phi_1$:

$$\nabla f(x)^T = e_N^T\ \Phi'_K(\bar{x}^{K-1})\ \cdots\ \Phi'_1(\bar{x}^0) \left(\begin{array}{c} I_n \\ O \end{array} \right), \tag{22}$$

où e_N est le N-ième vecteur de base de $\mathbb{R}^N$ et I_n est la matrice unité d'ordre n. Notons que la jacobienne de Φ_k s'écrit:

$$\Phi'_k = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ \frac{\partial \varphi_k}{\partial x_1} & \cdots & \cdots & \frac{\partial \varphi_k}{\partial x_{\mu_k}} & \cdots & \cdots & \frac{\partial \varphi_k}{\partial x_N} \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix},$$

où seule la μ_k-ième ligne de Φ'_k peut ne pas être formée de vecteurs de base de $\mathbb{R}^N$.

Le mode direct consiste à effectuer les multiplications matricielles de droite à gauche dans (22). En effet, les n premières lignes de la matrice de droite, qui forment la matrice identité I_n, sont obtenues dans la phase d'initialisation de DAD1. Partant de cette matrice, chaque étape k de DAD1 (**for** $k := 1$ **to** K) consiste à modifier la μ_k-ième ligne de la matrice $N \times n$ courante en la multipliant à gauche par Φ'_k.

Le mode inverse, quant à lui, consiste à faire les multiplications matricielles de gauche à droite dans (22). Voyons ce que cela donne. Définissons $(K+1)$ vecteurs $p^K, \ldots, p^0$ dans $\mathbb{R}^N$ comme suit:

$$p^K = e_N, \tag{23}$$

et pour $k = K-1, \ldots, 0$:

$$p^k = \left(\Phi'_{k+1}(\bar{x}^k)\right)^T \cdots \left(\Phi'_K(\bar{x}^{K-1})\right)^T e_N.$$

Alors, on a les formules de récurrence $p^k = \left(\Phi'_{k+1}(\bar{x}^k)\right)^T p^{k+1}$, pour $k = K-1, \ldots, 0$, c'est-à-dire:

$$p_i^k = \begin{cases} p_i^{k+1} + \dfrac{\partial \varphi_{k+1}}{\partial x_i}(\bar{x}^k)\, p_{\mu_{k+1}}^{k+1} & \text{si } i \neq \mu_{k+1} \\ \dfrac{\partial \varphi_{k+1}}{\partial x_{\mu_{k+1}}}(\bar{x}^k)\, p_{\mu_{k+1}}^{k+1} & \text{si } i = \mu_{k+1}, \end{cases} \tag{24}$$

Quant au gradient de f, on le récupère comme formant les n premières composantes de p^0:

$$\frac{\partial f}{\partial x_i} = p_i^0, \quad \text{pour} \quad 1 \leq i \leq n. \tag{25}$$

Remarquons que, sous l'hypothèse de consistance (3),

$$\frac{\partial f}{\partial x_i} \equiv p_i^0 = 0, \quad \text{pour} \quad n+1 \leq i \leq N. \tag{26}$$

C'est naturel, car alors f ne dépend pas de $x_{n+1}, \ldots, x_N$. Vérifions cela. Montrons d'abord que, pour les variables x_i telles que $\nu_i \geq 2$, on a (on utilise les notations de la section 2.1.2):

$$p_i^k = 0, \quad \text{pour} \quad 1 \leq k \leq \nu_i - 1.$$

Si $0 \leq k \leq \nu_i - 1$, alors $1 \leq k+1 \leq \nu_i$ et, par l'hypothèse de consistance (3), $i \notin P_{k+1}$. Donc $\partial\varphi_{k+1}/\partial x_i(\bar{x}^k) = 0$, pour $0 \leq k \leq \nu_i - 1$. On déduit alors de la première équation de (24) que $p_i^k = 0$ pour $k = \nu_i - 1$ ($\Leftrightarrow \quad i = \mu_{k+1}$) et de la seconde équation que $p_i^k = p_i^{k+1}$ pour $1 \leq k \leq \nu_i - 2$ ($i \neq \mu_{k+1}$). Par récurrence on obtient les relations ci-dessus. Ensuite, pour $n+1 \leq i \leq N$, on a $\partial\varphi_1/\partial x_i(\bar{x}_0) = 0$ (hypothèse de consistance). Alors, d'après (24) en $k = 0$, on a $p_i^0 = 0$, si $i = \mu_1$ et $p_i^0 = p_i^1$, si $i \neq \mu_1$. Mais dans ce dernier cas, $\nu_i \geq 2$ (x_i n'est pas la première variable modifiée) et donc $p_i^1 = 0$.

Nous verrons plus loin comment cet algorithme se ramène à l'algorithme DAD1 lorsque le modèle de programme est simple. Notons que cette approche montre clairement que la première méthode (mode direct) est plus coûteuse que la seconde (mode inverse) puisque, dans le premier cas, les K multiplications matricielles se font entre des matrices de dimension $N \times N$ et $N \times n$, au lieu de matrices de dimension $1 \times N$ et $N \times N$ dans le second cas.

2.3.6 Approche par substitutions régressives II

Nous généralisons ici au modèle étendu, l'approche par substitutions régressives décrite précédemment dans le cadre du modèle simple.

Comme plus haut, on s'intéresse à la variation de la variable x_N ($\equiv f$) lorsque la variable intermédiaire x_{μ_k} varie, k étant un indice fixé entre 1 et K. Remarquons qu'il faut cette fois spécifier l'indice k de l'instruction à laquelle la variable x_i a été modifiée ($i = \mu_k$), puisque celle-ci peut l'être par différentes instructions. On introduit alors pour $0 \leq k \leq K-1$, une fonction $\psi_k(\bar{x})$ définie par la partie du programme (2) située *après* l'instruction d'affectation d'ordre k:

$$\left[\begin{array}{l} \textbf{for} \quad i := k+1 \quad \textbf{to} \quad K \quad \textbf{do} \quad x_{\mu_i} := \varphi_i(\bar{x}); \\ \psi_k := x_N. \end{array}\right.$$

On définit également

$$\psi_K(\bar{x}) = x_N. \tag{27}$$

Alors, pour $k = 0, \ldots, K-1$, on a les formules de récurrence *régressives* suivantes:

$$\psi_k(\bar{x}) = \psi_{k+1}(x_1, \ldots, x_{\mu_{k+1}-1}, \varphi_{k+1}(\bar{x}), x_{\mu_{k+1}+1}, \ldots, x_N). \tag{28}$$

Celles-ci conduisent nécessairement à

$$\psi_0(\bar{x}) = f(x), \tag{29}$$

si le programme est bien construit dans le sens où la valeur de f ne dépend pas de l'initialisation des variables intermédiaires (hypothèse de consistance (3)), puisque dans ce cas, $\psi_0(\bar{x})$ est indépendant de $x_{n+1}, \ldots, x_N$.

On définit pour $1 \leq k \leq K$:

$$q_k \equiv \frac{\partial \psi_k}{\partial x_{\mu_k}}(\bar{x}^k) = \left[\nabla\psi_k(\bar{x}^k)\right]_{\mu_k}, \tag{30}$$

la composante μ_k du gradient de ψ_k par rapport à $\bar{x}$, évaluée en $\bar{x}^k$ (notation de la section 2.1.2). Cette quantité représente la variation de f lorsque l'on fait varier x_{μ_k} ($1 \leq \mu_k \leq N$) affecté à l'instruction d'ordre k. On associe donc un réel q_k à chaque instruction d'affectation

et non pas à chaque variable (ce qui n'est plus équivalent). On note encore $p^k \equiv \nabla\psi_k(\bar{x}^k)$, $0 \leq k \leq K$, donc $q_k = p^k_{\mu_k}$.

D'après (27), on obtient

$$p^K = e_N.$$

Pour $k = K-1, \ldots, 0$, les formules de récurrence régressives (28) donnent pour la i-ième composante p^k_i de p^k:

$$p^k_i = \frac{\partial \psi_k}{\partial x_i}(\bar{x}^k) = (1 - \delta_{i,\mu_{k+1}})\, p^{k+1}_i + \frac{\partial \varphi_{k+1}}{\partial x_i}(\bar{x}^k) p^{k+1}_{\mu_{k+1}},$$

ce qui conduit aux formules (24) et montre que les vecteurs $p^K, \ldots, p^0$ sont identiques à ceux définis précédemment. L'approche présente fournit toutefois une nouvelle interprétation de ces vecteurs. D'après (29), le gradient de f se retrouve encore dans les composantes de p^0 – comme en (25)-(26).

2.3.7 Approche par dualité II

Il est délicat d'étendre l'approche duale I à des fonctions φ_k pouvant dépendre également des variables $x_k, \ldots, x_N$. La raison est que la substitution "instruction d'affectation $\rightarrow$ équation" faite précédemment n'est plus valable. En effet, d'une part, cela fournit plus d'équations que d'inconnues (il faut au moins une instruction d'affectation pour définir chaque variable intermédiaire), ce qui pose déjà un problème de compatibilité. D'autre part, cette substitution n'est généralement pas licite. Par exemple, on ne peut pas remplacer l'instruction d'affectation $x_4 := x_4 + 2$ par l'équation $x_4 = x_4 + 2$! Il ne servirait à rien d'ailleurs de prendre comme hypothèse que dans $x_{\mu_k} := \varphi_k(\bar{x})$, φ_k ne dépende pas de la variable modifiée x_{μ_k}, car si une variable intermédiaire est modifiée deux fois, c'est en général pour lui donner la seconde fois une valeur différente de celle qu'elle avait reçue la première fois. Mais alors cette nouvelle valeur n'est plus compatible avec l'équation déduite de la première affectation. Comme exemple, considérons le cas suivant ($n = 1$ et $N = 3$) dont le système d'équations associé est incompatible:

$$\left[\begin{array}{l} x_2 := x_1; \\ x_3 := x_2; \\ x_2 := x_3 + 1. \end{array}\right.$$

Sans hypothèse supplémentaire, on s'en sort en liant par des équations, non pas les variables $\bar{x} = (x_1, \ldots, x_N)$, mais leurs *valeurs*: $\tilde{x} = (\bar{x}^0, \bar{x}^1 \ldots, \bar{x}^K) \in I\!R^{N\times(K+1)}$. En utilisant les fonctions Φ_k introduites dans l'approche de Speelpenning, les équations (ou contraintes) sont les suivantes:

$$\bar{x}^k = \Phi_k(\bar{x}^{k-1}) \in I\!R^N, \quad \text{pour} \quad k = 1, \ldots, K.$$

Les valeurs $\bar{x}^0_1, \ldots, \bar{x}^0_n$ sont données (ce sont les valeurs des variables indépendantes) et les valeurs $\bar{x}^0_{n+1}, \ldots, \bar{x}^0_N$ sont arbitraires (sous l'hypothèse de consistance (3), ces valeurs n'influencent pas le résultat).

Cette fois, la fonction dont on cherche le gradient est $\tilde{f}(\tilde{x}) = x^K_N$, les variables de contrôle sont les N valeurs $\bar{x}^0_1, \ldots, \bar{x}^0_N$ et les variables d'état sont les NK valeurs $\bar{x}^1, \ldots, \bar{x}^K$. Comme précédemment, on définit un lagrangien (même notation) en introduisant un multiplicateur,

$p^k \in I\!R^N$, $1 \leq k \leq K$, par contrainte:

$$\ell(\bar{x}^0, \bar{x}^1, \ldots, \bar{x}^K, p^1, \ldots, p^K) = \bar{x}_N^K - \sum_{k=1}^{K} \sum_{i=1}^{N} \left(\bar{x}_i^k - \left(\Phi_k(\bar{x}^{k-1}) \right)_i \right) p_i^k.$$

Pour obtenir les équations adjointes (en p^k), on annule la dérivée du lagrangien par rapport aux variables d'état. En dérivant par rapport à $\bar{x}_i^K$, $1 \leq i \leq N$, on obtient:

$$p_i^K = \delta_{iN}.$$

Autrement dit, $p^K = e_N$, le N-ième vecteur de base de $I\!R^N$: on retrouve (23). Considérons à présent le cas où $1 \leq k \leq K-1$. La dérivation du lagrangien par rapport à $\bar{x}_i^k$ donne:

$$\begin{aligned}
p_i^k &= \sum_{l=1}^{K} \sum_{j=1}^{N} \frac{\partial \left(\Phi_l(\bar{x}^{l-1}) \right)_j}{\partial \bar{x}_i^k} p_j^l \\
&= \sum_{j=1}^{N} \frac{\partial \left(\Phi_{k+1}(\bar{x}^k) \right)_j}{\partial \bar{x}_i^k} p_j^{k+1} \\
&= \sum_{j \neq \mu_{k+1}} \delta_{ij}\, p_j^{k+1} + \frac{\partial \varphi_{k+1}(\bar{x}^k)}{\partial x_i} p_{\mu_{k+1}}^{k+1} \\
&= (1 - \delta_{i,\mu_{k+1}})\, p_i^{k+1} + \frac{\partial \varphi_{k+1}(\bar{x}^k)}{\partial x_i} p_{\mu_{k+1}}^{k+1},
\end{aligned}$$

ce qui redonne les formules (24) pour $k = K-1, \ldots, 1$, permettant de trouver $p^{K-1}, \ldots, p^1$ de façon rétrograde.

Il reste à calculer le gradient de $f \equiv x_N^K$ dont la i-ième composante s'obtient en dérivant le lagrangien par rapport à $\bar{x}_i^0$, $1 \leq i \leq N$. Les formules sont identiques à celles de p^k avec $k = 0$ et il suffit de continuer la récurrence précédente jusqu'à $k = 0$. On retrouve finalement les composantes du gradient dans p^0.

2.3.8 L'algorithme DAI2

Du point de vue programmation, les formules (24) peuvent être implémentées en ne mémorisant qu'un seul vecteur $p \in I\!R^N$, initialisé à e_N (voir (23)) et mis à jour suivant les règles (24). Le gradient de f se récupère comme en (25). Cela donne l'algorithme suivant.

Phase progressive

$$\left[\begin{array}{l}
\textbf{for} \quad k := 1 \quad \textbf{to} \quad K \quad \textbf{do} \quad \{ \\
\qquad \varphi_{k,j} := \frac{\partial \varphi_k}{\partial x_j}, \quad \forall j \in P_k; \\
\qquad x_{\mu_k} := \varphi_k(x_{P_k}); \\
\} \\
f := x_N.
\end{array} \right. \tag{31}$$

Phase régressive

$$
\left[
\begin{array}{l}
\textbf{for}\ \ i := 1\ \ \textbf{to}\ \ N-1\ \ \textbf{do}\ \ p_i := 0; \\
p_N := 1; \\
\textbf{for}\ \ k := K\ \ \textbf{down to}\ \ 1\ \ \textbf{do}\ \ \{ \\
\quad p_j := p_j + \varphi_{k,j}\, p_{\mu_k}, \quad \forall j \in P_k \backslash \{\mu_k\}; \\
\quad p_{\mu_k} := \varphi_{k,\mu_k}\, p_{\mu_k}; \\
\} \\
\textbf{for}\ \ i := 1\ \ \textbf{to}\ \ n\ \ \textbf{do}\ \ \frac{\partial f}{\partial x_i} := p_i.
\end{array}
\right. \tag{32}
$$

Remarquons que cette fois-ci, il est important de calculer les $\varphi_{k,j}$ *avant* d'évaluer x_{μ_k}, puisque φ_k peut dépendre de x_{μ_k}.

A la fin de la boucle en k de la phase régressive, p contient le vecteur $p^0 = \nabla\psi_0(\bar{x}^0)$ dont les n premières composantes donnent le gradient de f. Les $N-n$ dernières composantes doivent être nulles si le programme est bien construit – voir (26). Une composante non nulle signifierait que f dépend de la valeur donnée initialement à une variable auxiliaire.

Dans le cas du modèle simple, on retrouve l'algorithme DAI1 avec $q \equiv p$ si $p_{\mu_k} = p_{n+k}$ n'est pas mis à zéro dans la phase régressive. Cela ne modifie pas la valeur calculée des p_i car dans le cas du modèle simple, $P_k \subset \{1, \ldots, n+k-1\}$.

2.3.9 Temps d'exécution et encombrement mémoire

Avec les notations précédentes, le temps $T_{\mathrm{DAI2}}(f, \nabla f)$ nécessaire au calcul simultané de f et de ∇f par l'algorithme DAI2 est estimé dans la proposition suivante (Griewank (1989)). On a la même estimation pour l'algorithme DAI1.

Proposition 2.3 *Le temps $T_{\mathrm{DAI2}}(f, \nabla f)$ nécessaire au calcul de f et de ∇f par l'algorithme DAI2 vérifie l'estimation suivante:*

$$
\frac{T_{\mathrm{DAI2}}(f, \nabla f)}{T_{(2)}(f)} \leq C'_{\mathcal{F}}, \tag{33}
$$

où $T_{(2)}(f)$ est le temps nécessaire au calcul de f par l'algorithme (2) et $C'_{\mathcal{F}}$ est une constante ne dépendant que des fonctions de la bibliothèque $\mathcal{F}$.

Preuve. L'examen de la phase régressive de l'algorithme DAI2, permet d'écrire:

$$
T_{\mathrm{DAI2}}(f, \nabla f) = \sum_{k=1}^{K} T_{\mathrm{DAI2}}(\varphi_k, \nabla\varphi_k).
$$

On ne compte pas dans T_{DAI2} le temps nécessaire à l'initialisation des variables duales p_i. Ensuite, en utilisant l'inégalité de Hölder, on obtient

$$
\begin{aligned}
T_{\mathrm{DAI2}}(f, \nabla f) &\leq \max_{1 \leq k \leq K} \left(\frac{T_{\mathrm{DAI2}}(\varphi_k, \nabla\varphi_k)}{T(\varphi_k)} \right) \sum_{k=1}^{K} T(\varphi_k) \\
&\leq \max_{\varphi \in \mathcal{F}} \left(\frac{T_{\mathrm{DAI2}}(\varphi, \nabla\varphi)}{T(\varphi)} \right) T_{(2)}(f) \\
&= C'_{\mathcal{F}}\, T_{(2)}(f),
\end{aligned}
$$

où la constante

$$C'_{\mathcal{F}} = \max_{\varphi \in \mathcal{F}} \left(\frac{T_{\mathrm{DAI2}}(\varphi, \nabla\varphi)}{T(\varphi)} \right)$$

ne dépend que de la classe $\mathcal{F}$ des fonctions intermédiaires. □

Il est remarquable que le majorant obtenu ne dépende que de la classe $\mathcal{F}$, et cela quelle que soit la complexité du programme calculant f. En particulier, le fait que cette constante soit indépendante de n est un des atouts majeurs du mode inverse de dérivation.

Comme Griewank (1989) et comme nous l'avons fait en section 2.2.4, on peut donner une estimation plus précise de la borne $C'_{\mathcal{F}}$. Rapidement, on voit que les opérations à effectuer avec le mode inverse sont pratiquement identiques à celles de l'algorithme (6), à ceci près. Il y a $|P_k| - 1$ additions dans l'algorithme (6) pour sommer les quantités $\varphi_{k,j} u_j$, alors qu'il y en a $|P_k|$ dans DAI2 pour ajouter $\varphi_{k,j} p_{\mu_k}$ à p_j (on considère le cas le plus défavorable où $\mu_k \notin P_k$). On peut donc s'attendre à ce que les estimations de $C'_{\mathcal{F}}$ soient celles de $C_{\mathcal{F}}$ majorées d'une unité. De façon plus détaillée, si on se restreint à la classe $\mathcal{F}_0$ des fonctions arithmétiques élémentaires, on trouve pour l'algorithme DAI1

$$C'_{\mathcal{F}_0} \leq \max \left(3, 2 + \frac{T(+)}{T(-)}, 3 + \frac{2T(+)}{T(*)}, 2 + \frac{T(+) + T(-) + T(*)}{T(/)} \right),$$

où les arguments du max correspondent successivement aux opérations +, -, * et /. En effet, par exemple, les opérations de la phase rétrograde de DAI1 correspondant à la division $x_k := x_i / x_j$ s'écrivent: $p_i := p_i + p_k / x_j$ et $p_j := p_j - x_k * p_k / x_j$. Sa contribution à $C'_{\mathcal{F}_0}$ est donc (on évalue une seule fois p_k / x_j):

$$\frac{5T(\boxed{\rightarrow}) + 3T(\boxed{-}) + T(+) + T(-) + T(*) + 2T(/)}{2T(\boxed{\rightarrow}) + T(\boxed{-}) + T(/)} \leq \max \left(3, 2 + \frac{T(+) + T(-) + T(*)}{T(/)} \right),$$

où, comme précédemment, $T(\boxed{\rightarrow})$ et $T(\boxed{-})$ représentent les temps qu'il faut pour aller chercher et pour sauvegarder un nombre flottant en mémoire. Avec l'algorithme DAI2, il y a en plus l'instruction $p_k := 0$ à exécuter (c'est une sauvegarde $\boxed{-}$), si bien que l'on a

$$C'_{\mathcal{F}_0} \leq \max \left(4, 2 + \frac{T(+)}{T(-)}, 3 + \frac{2T(+)}{T(*)}, 2 + \frac{T(+) + T(-) + T(*)}{T(/)} \right).$$

En faisant des hypothèses analogues à celles que nous avons faites en section 2.2.4, on a pour DAI1 et DAI2, l'estimation

$$C'_{\mathcal{F}_0} \leq 5.$$

De même, en considérant les opérations de la classe $\mathcal{F}_{\mathrm{F}}$ du FORTRAN et en faisant des hypothèses raisonnables sur les temps d'exécution relatifs des différentes opérations de $\mathcal{F}_{\mathrm{F}}$, on trouve encore

$$C'_{\mathcal{F}_{\mathrm{F}}} \leq 5. \tag{34}$$

Ces bornes ne tiennent compte que des coûts relatifs au calcul représenté par les algorithmes DAI1 et DAI2. Le surcoût occasionné par les opérations nécessaires à la mise en oeuvre informatique du mode inverse (comme la construction éventuelle d'un graphe de calcul) n'est pas repris dans ces nombres, si bien qu'en pratique la majoration (34) pourra être dépassée pour certaines implémentations.

L'algorithme DAI2 pris tel quel demande la mémorisation des dérivées partielles $\varphi_{k,j} = \partial\varphi_k/\partial x_j$ au cours de la phase progressive en vue de leur utilisation dans la phase régressive. Comme il faut également prévoir une case mémoire par variable duale p_i, l'espace mémoire *supplémentaire* requis sera de l'ordre de $\mathcal{O}(E)+\mathcal{O}(N)$, où E ($\geq K$) est le nombre d'opérations *élémentaires* effectuées (si toutes les fonctions intermédiaires sont élémentaires, on a $E = K$).

Cette estimation de l'espace mémoire est inquiétante, mais pessimiste. Elle dépend en effet fortement de l'implémentation. Par exemple, dans le cas du modèle simple et d'une représentation du programme par un graphe de calcul, il ne faut mémoriser aucune de ces dérivées partielles. Leur valeur peut en effet se retrouver aisément à partir d'objets déjà mémorisés: l'opération effectuée par φ_k et de ses opérandes: pour une addition, $\varphi_{k,j} = 1$, pour une soustraction $\varphi_{k,j} = \pm 1$, pour une multiplication $\varphi_k(x_{j_1}, x_{j_2}) = x_{j_1} * x_{j_2}$, $\varphi_{k,j_1} = x_{j_2}$, etc .... Mais s'il ne faut pas mémoriser les $\varphi_{k,j}$, il faut mémoriser la graphe de calcul, qui peut être plus encombrant encore.

Nous verrons plus loin une implémentation du mode inverse (l'implémentation *par codage de l'adjoint*) permettant de ne devoir mémoriser que les dérivées partielles des fonctions intermédiaires *non linéaires*. D'autres techniques, utilisables pour toutes implémentations, permettent de réduire l'espace mémoire au prix d'un accroissement du temps de calcul. Voir par exemple Griewank [25].

2.4 Extensions diverses

2.4.1 Calcul de jacobiennes

Supposons à présent que f soit à valeurs vectorielles ($m \geq 1$):

$$f : I\!R^n \to I\!R^m.$$

Calculer sa dérivée $f'(x)$ en un point $x \in I\!R^n$ revient à calculer sa matrice jacobienne que nous noterons également $f'(x)$, son élément (i, j) s'écrivant:

$$(f'(x))_{ij} = \frac{\partial f_i}{\partial x_j}.$$

Comme nous allons le voir, l'extension des deux modes de différentiation automatique, direct et inverse, à de telles fonctions est immédiat.

Comme pour les fonctions scalaires, on supposera que le programme calculant f a la forme suivante:

$$\left[\begin{array}{l} \textbf{for}\ \ k := 1\ \ \textbf{to}\ \ K\ \ \textbf{do}\ \ x_{\mu_k} := \varphi_k(x_{P_k}); \\ f := (x_{N-m+1}, \ldots, x_N), \end{array}\right. \tag{35}$$

On a convenu de numéroter les variables de telle sorte que les m composantes de f correspondent aux m dernières variables $x_{N-m+1}, \ldots, x_N$. Comme précédemment, on dira que ce modèle de programme est *simple* si $\mu_k = n + k$ et si l'ensemble d'indices P_k vérifie $P_k \subset \{1, \ldots, n + k - 1\}$.

Dans le cas de fonction scalaire, nous avons vu que le mode direct était bien adapté au calcul des dérivées directionnelles. Il en est de même ici. Ce mode de différentiation s'utilisera donc pour calculer $f'(x) \cdot u$, pour $u \in I\!R^n$ donné. L'adaptation de l'algorithme (6)

au programme (35), s'écrit:

$$\left[\begin{array}{l} \textbf{for}\ \ k := 1\ \ \textbf{to}\ \ K\ \ \textbf{do}\ \ \{ \\ \quad u_{\mu_k} := \sum_{j \in P_k}\ \varphi_{k,j}\, u_j; \\ \quad x_{\mu_k} := \varphi_k(x_{P_k}); \\ \} \\ f'(x) \cdot u := (u_{N-m+1}, \ldots, u_N). \end{array}\right.$$

Le calcul de la jacobienne $f'(x)$ pourra s'obtenir *colonne par colonne*. La colonne j de $f'(x)$, $f'(x) \cdot e_j$, s'obtiendra en prenant $u = e_j$ dans l'algorithme précédent. Le bilan de cette méthode s'établit comme suit. En ce qui concerne le temps de calcul, il sera en $\mathcal{O}(nT(f))$, puisque le programme précédent doit être exécuté pour chacune des n colonnes de la jacobienne et que, comme pour les fonctions scalaires, son coût en temps de calcul est de l'ordre de celui du calcul de f. Quant à l'espace mémoire supplémentaire requis, il sera en $\mathcal{O}(N)$ (stockage du vecteur auxiliaire u).

Nous avons vu que le mode inverse est bien adapté au calcul du gradient de fonctions scalaires. On pourra donc l'utiliser pour calculer le gradient de $x \in \mathbb{R}^n \to v^T f(x) \in \mathbb{R}$, où $v \in \mathbb{R}^m$ est donné. Les deux phases de l'algorithme DAI2, deviennent:

Phase progressive

$$\left[\begin{array}{l} \textbf{for}\ \ k := 1\ \ \textbf{to}\ \ K\ \ \textbf{do}\ \ \{ \\ \quad \varphi_{k,j} := \frac{\partial \varphi_k}{\partial x_j},\ \ \forall j \in P_k; \\ \quad x_{\mu_k} := \varphi_k(x_{P_k}); \\ \} \\ f := (x_{N-m+1}, \ldots, x_N). \end{array}\right. \tag{36}$$

Phase régressive

$$\left[\begin{array}{l} \textbf{for}\ \ i := 1\ \ \textbf{to}\ \ N-m\ \ \textbf{do}\ \ p_i := 0; \\ \textbf{for}\ \ i := 1\ \ \textbf{to}\ \ m\ \ \textbf{do}\ \ p_{N-m+i} := v_i; \\ \textbf{for}\ \ k := K\ \ \textbf{down to}\ \ 1\ \ \textbf{do} \\ \quad \textbf{if}\ \ p_{\mu_k} \neq 0\ \ \textbf{then}\ \ \{ \\ \qquad p_j := p_j + \varphi_{k,j}\, p_{\mu_k},\ \ \forall j \in P_k \backslash \{\mu_k\}; \\ \qquad p_{\mu_k} := \varphi_{k,\mu_k}\, p_{\mu_k}; \\ \quad \} \\ \nabla(v^T f) := (p_1, \ldots, p_n). \end{array}\right. \tag{37}$$

En fait, dans (36), nous n'avons pas écrit l'instruction donnant $v^T f(x) := \sum_{i=1}^m v_i f_i(x)$, mais nous en avons tenu compte en modifiant la valeur initiale de p dans la phase régressive. Le calcul de la jacobienne $f'(x)$ par le mode inverse se fera donc *ligne par ligne*, en faisant suivre la phase progressive de m phases régressives où l'initialisation de p correspondra à $v = e_j$, $1 \leq j \leq m$. Dans ce cas, la condition $p_{\mu_k} \neq 0$ de la phase régressive permettra de ne parcourir que la partie du calcul qui influence la valeur de la composante j de f. Cette fois, le temps de calcul sera en $\mathcal{O}(mT(f))$. L'espace mémoire supplémentaire requis reste le même que pour les fonctions scalaires: il est de l'ordre de $\mathcal{O}(E) + \mathcal{O}(N)$, où E est le nombre d'opérations élémentaires effectuées.

Sans information supplémentaire sur la 'nature' de la fonction à différentier, le choix du mode de différentiation pourra se faire en comparant les valeurs respectives de n et m. Si $n \ll m$, on utilisera le mode direct et si $n \gg m$, on utilisera le mode inverse. Le cas fréquent où $n \simeq m$ (résolution d'une équation non linéaire) est plus délicat. Le mode direct (sous la forme DAD2 ci-dessus) est plus économe en place mémoire si le graphe de calcul n'est pas formé (variante DAD3), puisque le mode inverse demande toujours la mémorisation des opérations effectuées durant la première phase. En ce qui concerne le temps de calcul, les choses sont moins claires. Si le mode direct est utilisé sous la forme DAD2 ci-dessus, le temps de calcul sera clairement supérieur à n fois le temps de calcul de f. Maintenant, si le graphe de calcul est mémorisé, tout dépendra de la structure de la fonction: si chaque variable indépendante influence peu de variables intermédiaires, le mode direct (variante DAD3) sera intéressant. Tandis que si le calcul de chaque composante de f se fait de façon relativement indépendante, le mode inverse (sous la forme ci-dessus) sera intéressant. Le cas limite est lorsque chaque composante de f est calculée par un programme différent; alors le mode inverse permet d'obtenir une estimation du type (33) sur le temps de calcul.

Dans beaucoup de problèmes de grande taille, la jacobienne que l'on veut calculer est *creuse*: structurellement, elle contient beaucoup de zéros. Dans ce cas, elle pourra s'obtenir en calculant moins que n dérivées directionnelles $f'(x) \cdot u$ (par le mode direct) ou moins que m gradients $\nabla(v^T f(x))$ (par le mode inverse). En effet, $f'(x)$ pourra être reconstituée en calculant $f'(x) \cdot u$ (resp. $\nabla(v^T f(x))$) pour quelques vecteurs u (resp. v) bien choisis. Par exemple, si $f'(x)$ est tridiagonale ($m = n$), il suffira de choisir 3 vecteurs, quel que soit l'ordre de la matrice $f'(x)$. Comme point d'entrée sur ces questions, nous renvoyons le lecteur aux articles de Coleman et Moré [8] et [9].

D'une façon générale, si la fonction à différentier est à valeurs vectorielles, il se peut très bien qu'aucun des modes direct ou inverse ne soit optimal et la meilleure manière de procéder dépendra de la fonction considérée. On pourrait alors songer à minimiser le nombre d'opérations à effectuer en examinant le graphe de calcul dès que celui-ci est construit. De toute façon, pour qu'une telle méthode soit acceptable il faut que le programme de calcul de f soit exempt d'instructions conditionnelles qui feraient que ce graphe changerait avec la valeur donnée aux variables indépendantes. Dans les cas favorables, le calcul de dérivées se déroulerait en trois étapes:

(i) constitution du graphe de calcul,

(ii) optimisation des opérations à effectuer pour le calcul des dérivées et

(iii) calcul effectif d'une dérivée pour une valeur particulière des variables indépendantes.

Comme en l'absence d'instructions conditionnelles "gênantes", les deux premières étapes ne dépendent pas du point où est calculée la dérivée, elles peuvent être faites une fois pour toutes. Seule la troisième étape, rendue efficace par l'étape (ii), serait à adapter au point courant. Toutefois, il est conjecturé – voir Griewank [24] – que le problème d'optimisation de l'étape (ii) est NP *hard* (comme point d'entrée sur ces questions, on pourra consulter le livre de Garey et Johnson (1979)).

Supposons à présent que $m = n$. On vient de voir que les modes direct et inverse permettent d'évaluer respectivement les n éléments d'une colonne ou d'une ligne de $f'(x)$, en un temps *relatif* indépendant de n. On peut alors se demander, comme Nesterov (1991), s'il n'existerait pas un algorithme, disons DIAG, permettant d'évaluer les n éléments de la

diagonale de $f'(x)$ en un temps relatif indépendant de n, c'est-à-dire avec l'estimation:

$$\frac{T_{\mathrm{DIAG}}(\mathrm{diag}\, f'(x))}{T(f)} \leq C. \tag{38}$$

Cette question est importante car, par exemple, pour beaucoup de méthodes numériques la diagonale de $f'(x)$ peut être utilisée comme préconditionneur. Une indication sur cette question est donnée par la proposition suivante.

Proposition 2.4 *Supposons qu'il existe un algorithme, disons DIAG, calculant la diagonale de la jacobienne d'une fonction $f : \mathbb{R}^n \to \mathbb{R}^n$ en un temps $T_{\mathrm{DIAG}}(\mathrm{diag}\, f'(x))$ satisfaisant (38), où C est une constante indépendante de n. Alors, il existe un algorithme, disons JAC, calculant la jacobienne de f en un temps $T_{\mathrm{JAC}}(f'(x))$ satisfaisant*

$$T_{\mathrm{JAC}}(f'(x)) \leq C\, \left(n^2 + T(f)\right). \tag{39}$$

Cet algorithme permettrait de calculer le produit de deux matrices d'ordre n en $\mathcal{O}(n^2)$ opérations.

Preuve. Introduisons la fonction

$$F : \mathbb{R}^{n^2} \to \mathbb{R}^{n^2} : z \to F(z),$$

obtenue à partir de f par composition

$$F = \psi \circ f \circ \varphi,$$

où $\varphi : \mathbb{R}^{n^2} \to \mathbb{R}^n : z \to \varphi(z)$ et $\psi : \mathbb{R}^n \to \mathbb{R}^{n^2} : y \to \psi(y)$ sont définies par

$$\varphi_j(z) = \sum_{k=1}^{n} z_{(j-1)n+k}, \quad 1 \leq j \leq n,$$

$$\psi_l(y) = y_{((l-1) \bmod n)+1}, \quad 1 \leq l \leq n^2.$$

Alors, pour tout indice $i \in \{1, \ldots, n^2\}$, on a

$$\begin{aligned} \frac{\partial F_i}{\partial z_i}(z) &= \frac{\partial \left(f_{((i-1) \bmod n)+1} \circ \varphi\right)}{\partial z_i}(z) \\ &= \sum_{j=1}^{n} \frac{\partial f_{((i-1) \bmod n)+1}}{\partial x_j} \frac{\partial \varphi_j}{\partial z_i}(z). \end{aligned}$$

En prenant $i = (q-1)n + p$, avec $1 \leq q \leq n$ et $1 \leq p \leq n$, on a

$$((i-1) \bmod n) + 1 = p,$$

et

$$\frac{\partial \varphi_j}{\partial z_i}(z) = \begin{cases} 1 & \text{si } \ j = q \\ 0 & \text{sinon.} \end{cases}$$

Par conséquent, on obtient

$$\frac{\partial F_i}{\partial z_i}(z) = \frac{\partial f_p}{\partial x_q}(\varphi(z)).$$

Lorsque $i = (q-1)n + p$ parcourt $\{1, \ldots, n^2\}$, p et q parcourent $\{1, \ldots, n\}$. Dès lors, lorsqu'on calcule la diagonale de la jacobienne $F'(z)$ au moyen de l'algorithme DIAG, on calcule en fait les n^2 éléments de la jacobienne $f'(\varphi(z))$ de f. De plus, pour $x \in \mathbb{R}^n$ donné, on peut facilement trouver $z \in \mathbb{R}^{n^2}$ tel que $\varphi(z) = x$: par exemple, on peut prendre $z_{(q-1)n+p} = \delta_{p1} x_q$, où δ est le symbole de Kronecker. Ceci implique que

$$T(f') = T_{\mathrm{DIAG}}(\mathrm{diag}\, F').$$

Par hypothèse, on a $T_{\mathrm{DIAG}}(\mathrm{diag}\, F'(z)) \leq C\, T(F)$. D'autre part, on peut manifestement trouver des algorithmes tels que $T(F) = T(\varphi) + T(f) + T(\psi)$, $T(\varphi) = n^2 T(+)$ et $T(\psi) = n^2 T(+)$. On en déduit (39).

La dernière partie de la proposition s'obtient en particularisant $f : \mathbb{R}^n \to \mathbb{R}^n$, comme Morgenstern (1985). On prend

$$f(x) = ABx,$$

où A et B sont deux matrices d'ordre n. Bien sûr, la jacobienne de f est le produit matriciel AB et on a l'estimation $T(f) \leq C\, n^2$, si on calcule $f(x)$ en réalisant le produit $u = Bx$ en $\mathcal{O}(n^2)$ opérations suivi du produit Au également en $\mathcal{O}(n^2)$ opérations. Alors (39) conduit au résultat. □

Notons que, le meilleur algorithme (ω minimal) connu de Strassen (1990), réalisant le produit de deux matrices d'ordre n en $\mathcal{O}(n^\omega)$ opérations, demande $\omega = 2.50$.

2.4.2 Différentiation d'ordre supérieur

Le calcul des dérivées d'ordre supérieur est utile dans de nombreuses situations. Comme exemple important, citons celui du calcul de la direction de Newton d_N pour la minimisation d'une fonction scalaire $f : \mathbb{R}^n \to \mathbb{R}$. Celle-ci est définie par le système *linéaire* suivant:

$$\nabla^2 f(x)\, d_N = -\nabla f(x),$$

qui fait intervenir les dérivées secondes de f. L'utilisation de certaines méthodes itératives pour résoudre ce système ne demande que le calcul de $\nabla^2 f(x)\, u$, pour différents vecteurs $u \in \mathbb{R}^n$. La méthode du gradient conjugué, que l'on interrompt lorsque l'on détecte le caractère indéfini éventuel du hessien, en est un exemple: cette combinaison de la méthode de Newton et du gradient conjugué porte le nom de *méthode de Newton tronquée* [56, 11, 53]. Ces méthodes sont attrayantes parce que les quantités $\nabla^2 f(x)\, u$ peuvent être évaluées en un temps relatif (au temps de calcul de f) indépendant de n. Pour voir cela, remarquons que $\nabla^2 f(x)\, u$ peut s'obtenir en évaluant le gradient de l'application $x \to \delta(x) \equiv \nabla f(x)^T u = f'(x) \cdot u$ et que cela peut se faire en utilisant le mode direct pour évaluer l'application δ et le mode inverse pour évaluer son gradient. On peut aussi l'obtenir en prenant la dérivée directionnelle suivant u (mode direct) de l'application gradient $x \to \nabla f(x)$ (mode inverse). Dans cette section, nous allons voir comment généraliser cela à des ordres de dérivation directionnelle supérieurs.

Avant d'entrer dans le vif du sujet, il nous faut préciser quelques notations. Soient x un point de $\mathbb{R}^n$ et ξ une fonction réelle, définie et régulière dans un voisinage de x. Pour $t \geq 1$ entier, donnons-nous t directions $u^1, \ldots, u^t$ de $\mathbb{R}^n$ et désignons par

$$\xi^{(t)}(x) \cdot (u^1, \ldots, u^t) \in \mathbb{R}, \tag{40}$$

la dérivée *multidirectionnelle* d'ordre t de ξ en x dans la multidirection $(u^1, \ldots, u^t)$. Nous nous intéressons donc au calcul des dérivées multidirectionnelles d'ordre t de $f : \mathbb{R}^n \to \mathbb{R}^m$:

$$f^{(t)}(x) \cdot (u^1, \ldots, u^t) \in \mathbb{R}^m,$$

dont chaque composante est donnée par la formule (40) avec $\xi = f_j$, $1 \leq j \leq m$.

On obtient les dérivées *unidirectionelles* d'ordre t de ξ dans la direction $u \in \mathbb{R}^n$ en prenant $u^1 = \cdots = u^t = u$ dans (40). Cette quantité se notera

$$\xi^{(t)} \equiv \xi^{(t)}(x) \cdot (u)^t. \tag{41}$$

On sera aussi amené à considérer le $(t+1)$-ième terme du développement de Taylor de $\xi(x+u)$ autour de x, que l'on notera

$$\xi^{[t]} \equiv \frac{1}{t!}\, \xi^{(t)}(x) \cdot (u)^t. \tag{42}$$

En pratique, il est beaucoup plus simple d'évaluer (41) que (40). En effet, comme on le montre en annexe A, le calcul de (40) demande en général l'évaluation *et la mémorisation* de 2^t valeurs pour chaque variable du code (croissance exponentielle), alors qu'avec la formule (41), il suffira de calculer et de mémoriser $t+1$ valeurs par variable du code. Notons que si c'est réellement *une* dérivée multidirectionnelle qui doit être calculée, on peut toujours l'obtenir à partir des dérivées unidirectionnelles, mais la seule formule que nous connaissions (voir annexe A) demande l'évaluation de 2^{t-1} dérivées unidirectionnelles, pour le calcul d'*une* dérivée t-directionnelle. Certains auteurs s'intéressent également au calcul de toutes les dérivées partielles d'ordre inférieur ou égal à t: voir Neidinger (1989) et sa bibliographie.

D'autre part, il est aussi préférable d'évaluer le terme de Taylor (42) plutôt que (41). La raison vient de ce que les formules de transformation des termes de Taylor ne font pas intervenir de factorielles, contrairement aux formules correspondantes pour (41). Ceci est avantageux car l'évaluation continuelle de ces factorielles est coûteuse et peut donner lieu à des "overflows".

Comme pour le calcul des dérivées premières, on distingue un mode direct et un mode inverse de dérivation. Afin d'éviter certaines difficultés techniques, nous supposerons que la fonction est représentée par un modèle simple de programme.

Le mode direct

Le mode direct pour le calcul des termes de la série de Taylor, peut être introduit en suivant le principe développé précédemment pour le calcul des dérivées premières en mode direct – voir par exemple Rall (1981). Cette fois, au lieu de ne générer que les variables intermédiaires et leur dérivée directionnelle, on génère en une seule passe les $(t+1)$ premiers termes de leur série de Taylor.

Pour une direction u donnée dans $\mathbb{R}^n$, on associe à chaque variable x_i, le $(t+1)$-uple suivant:

$$[x_i] = (x_i^{[0]}, x_i^{[1]}, \ldots, x_i^{[t]}), \quad \text{pour } 1 \leq i \leq N,$$

où $x_i^{[j]}$ est donné par (42) avec $\xi = x_i$. Pour les variables indépendantes, ce $(t+1)$-uple s'écrit simplement

$$[x_i] = (x_i, u_i, 0, \ldots, 0), \quad \text{pour } 1 \leq i \leq n, \tag{43}$$

où u_i est la i-ième composante de u. Ensuite, ces quantités sont évaluées pas à pas, au fur et à mesure de la transformation des variables. Pour obtenir le programme final on transforme chaque ligne $x_k := \varphi_k(x_{P_k})$ du programme (35) en la ligne $[x_k] := [\varphi]_k([x_{P_k}])$, où $[\varphi]_k$ est une fonction à définir et $[x_{P_k}]$ est la forme abrégée de $\{[x_i] : i \in P_k\}$. Ceci conduit au programme suivant:

$$\left[\begin{array}{l} \textbf{for} \quad i := 1 \quad \textbf{to} \quad n \quad \textbf{do} \quad [x_i] := (x_i, u_i, 0, \ldots, 0); \\ \textbf{for} \quad k := n+1 \quad \textbf{to} \quad N \quad \textbf{do} \quad [x_k] := [\varphi]_k([x_{P_k}]); \\ \textbf{for} \quad j := 1 \quad \textbf{to} \quad m \quad \textbf{do} \quad [f_j] := [x_{N-m+j}]. \end{array}\right. \tag{44}$$

Il reste à préciser comment sont déterminées les fonctions

$$[\varphi]_k : I\!R^{|P_k|(t+1)} \to I\!R^{(t+1)} : \{[x_i] : i \in P_k\} \to [x_k],$$

à partir des fonctions $\varphi_k : I\!R^{|P_k|} \to I\!R$.

Remarquons d'abord que, lorsque $t = 1$, la variante (6) de l'algorithme DAD2 s'écrit sous la forme ci-dessus (u_j devient $x_j^{[1]}$) avec $[\varphi]_k$ définie par

$$[\varphi]_k([x_{P_k}]) \equiv \left(x_k^{[0]}, x_k^{[1]}\right) = \left(\varphi_k(x_{P_k}^{[0]}), \sum_{j \in P_k} \frac{\partial \varphi_k}{\partial x_j}(x_{P_k}^{[0]})\, x_j^{[1]}\right).$$

Il s'agit de voir comment généraliser cela à des ordres de dérivation supérieurs.

En supposant que toutes les fonctions intermédiaires φ_k peuvent être décomposées en fonctions élémentaires, le problème revient à voir comment ces fonctions transforment les séries de Taylor. A titre d'exemple, considérons le cas du produit de deux variables:

$$x_k = x_i\, x_j.$$

On a

$$\begin{aligned} x_k' &= x_i'\, x_j + x_i\, x_j', \\ x_k'' &= x_i''\, x_j + 2 x_i'\, x_j' + x_i\, x_j'', \end{aligned}$$

où $x_i' \equiv x_i'(x) \cdot u$ et $x_i'' \equiv x_i''(x) \cdot (u)^2$. Si on note $x_i^{(t)} \equiv x_i^{(t)}(x) \cdot (u)^t$, on obtient par récurrence (voir annexe A):

$$x_k^{(t)} = \sum_{l=0}^{t} \binom{t}{l} x_i^{(l)}\, x_j^{(t-l)},$$

où $\binom{t}{l} = \frac{t!}{l!(t-l)!}$. On en déduit la formule

$$x_k^{[t]} = \sum_{l=0}^{t} x_i^{[l]}\, x_j^{[t-l]}.$$

Cette formule permet de déterminer les termes de la série de Taylor de x_k à partir de ceux de x_i et x_j.

De la même manière, on peut établir des formules de ce genre pour toutes les opérations élémentaires usuelles. Nous en avons rassemblé quelques-unes en annexe A (Table 6). Ces formules appellent les commentaires suivants:

- Comme nous l'annoncions au début, ces formules ne contiennent pas de factorielles.
- Le calcul du terme d'ordre t du développement de x_k peut demander la connaissance des termes d'ordre $l = 0, \ldots, t$ du développement des variables dont x_k dépend (c'est le cas pour la formule du produit). Il est donc généralement nécessaire de générer tous les éléments des $(t+1)$-uples $[x_i]$, et pas seulement x_i et $x_i^{[t]}$.
- Ce calcul peut aussi demander l'évaluation des termes de Taylor de variables auxiliaires supplémentaires. C'est le cas de la fonction 'sinus' dont l'évaluation des termes de Taylor demande le calcul simultané des développements du 'sinus' et du 'cosinus'.
- Compte tenu des deux points précédents, le nombre d'opérations à effectuer pour la transformation des termes d'ordre inférieur ou égal à t par une opération élémentaire reste de l'ordre de $\mathcal{O}(t^2)$.

Maintenant, rien n'empêche de donner en entrée à l'algorithme (44) des $(t+1)$-uples $[x_i]$ quelconques à la place des $(t+1)$-uples (43). On obtient alors le programme suivant:

$$\left[\begin{array}{l} \textbf{for} \quad i := 1 \quad \textbf{to} \quad n \quad \textbf{do} \quad [x_i] := (x_i^{[0]}, x_i^{[1]}, \ldots, x_i^{[t]}); \\ \textbf{for} \quad k := n+1 \quad \textbf{to} \quad N \quad \textbf{do} \quad [x_k] := [\varphi]_k([x_{P_k}]); \\ \textbf{for} \quad j := 1 \quad \textbf{to} \quad m \quad \textbf{do} \quad [f_j] := [x_{N-m+j}]. \end{array}\right. \tag{45}$$

Celui-ci peut être interprété comme suit. Pour $1 \leq i \leq n$, $[x_i]$ peut être vu comme les $(t+1)$ premiers termes d'un développement de Taylor autour d'un point α_0 de la i-ième composante d'une application

$$T : I\!R^r \to I\!R^n : \alpha \to x(\alpha).$$

Alors, en sortie du programme (45), $[f_j]$ donne les $(t+1)$ premiers termes du développement de Taylor autour du même point α_0 de la composante j de l'application composée

$$f \circ T : I\!R^r \to I\!R^m : \alpha \to f(x(\alpha)).$$

Par exemple, ce programme peut servir à donner une description à l'ordre t de la manière dont une courbe de $I\!R^n$

$$\alpha \in I\!R \to x(\alpha) \equiv x + \sum_{l=1}^{t} \frac{\alpha^l}{l!} x^{(l)} + \mathcal{O}(\alpha^{t+1}) \in I\!R^n$$

est transformée par l'application f. Les valeurs x et $x^{(l)} \equiv x^{(l)}(0)$, $1 \leq l \leq t$, étant connues, il suffit de prendre $x^{[0]} = x$ et $x^{[l]} = \frac{\alpha^l}{l!} x^{(l)}$ (α quelconque), $1 \leq l \leq t$, en entrée du programme (45). Alors, si $f^{[l]} \in I\!R^m$, $0 \leq l \leq t$, sont les valeurs récupérées en sortie, on a

$$f(x(\alpha)) = \sum_{l=0}^{t} \frac{\alpha^l}{l!} \left(\frac{l!}{\alpha^l} f^{[l]} \right) + \mathcal{O}(\alpha^{t+1}),$$

où $\frac{l!}{\alpha^l} f^{[l]}$ est indépendant de α (puisque $f^{[l]}$ est homogène de degré l en α) et vaut $(f \circ x)^{(l)}(0)$.

L'évaluation des $(t+1)$ premiers termes de Taylor par le mode direct se fera donc en une seule passe avec un temps calcul en $\mathcal{O}(t^2)$. L'espace mémoire *supplémentaire* sera en $\mathcal{O}(Nt)$: pour chaque variable x_i du programme original, il faudra mémoriser $[x_i] \in I\!R^{t+1}$.

Le mode inverse

Un mode inverse peut être introduit de la manière suivante – voir Griewank [23]. Le programme (45) définit une application

$$\{[x_i] : 1 \leq i \leq n\} \in I\!R^{n(t+1)} \to \{[f_j] : 1 \leq j \leq m\} \in I\!R^{m(t+1)}.$$

Pour $v \in \mathbb{R}^m$ donné, cette application permet de considérer les quantités

$$v^T f^{[s]} \equiv \sum_{j=1}^{m} v_j f_j^{[s]} \in \mathbb{R}, \quad 0 \leq s \leq t,$$

comme des fonctions de $x_i^{[r]}$, $1 \leq i \leq n$, $0 \leq r \leq t$. La dépendance de $v^T f^{[s]}$ en $x_i^{[r]}$ n'est pas arbitraire. Par exemple

$$\frac{\partial (v^T f^{[s]})}{\partial x_i^{[r]}} = 0, \quad \text{si } 0 \leq s < r,$$

puisque le s-ième terme du développement de Taylor de f ne dépend que des termes d'ordre $\leq s$ du développement de $x = T(\alpha)$. De plus, à partir de la formule donnant la transformation des termes du développement de Taylor par composition (Schwartz [62, Chapitre III, Théorème 21*ter*]), on peut montrer que

$$\frac{\partial (v^T f^{[s]})}{\partial x_i^{[r]}} = \frac{\partial (v^T f^{[s-r]})}{\partial x_i^{[0]}}, \quad \text{si } 0 \leq r \leq s.$$

Il est donc naturel de ne s'intéresser qu'aux dérivées par rapport à $x^{[0]} \equiv x$ et en particulier aux dérivées du terme d'ordre maximal

$$\frac{\partial (v^T f^{[t]})}{\partial x_i} = \frac{1}{t!} v^T f^{(t+1)}(x) \cdot (e_i, u, \ldots, u).$$

Le programme représentant cette application s'obtient à partir de (45). Il s'écrit:

$$\left[\begin{array}{l} \textbf{for} \quad i := 1 \quad \textbf{to} \quad n \quad \textbf{do} \quad [x_i] := (x_i^{[0]}, x_i^{[1]}, \ldots, x_i^{[t]}); \\ \textbf{for} \quad k := n+1 \quad \textbf{to} \quad N \quad \textbf{do} \quad [x_k] := [\varphi]_k([x_{P_k}]); \\ \textbf{return} \ (\textstyle\sum_{j=1}^{m} v_j x_{N-m+j}^{[t]}). \end{array} \right. \tag{46}$$

Le gradient de la fonction réalisée par ce programme peut alors s'obtenir par les techniques développées dans la section 2.3. Voyons cela. L'instruction $[x_k] := [\varphi]_k([x_{P_k}])$ se décompose en une suite d'instructions exécutées dans l'ordre suivant:

$$\begin{array}{lcll} x_k^{[0]} & := & [\varphi]_k^0(x_{P_k}^{[0]}); & // \quad x_i^{[0]} \equiv x_i, \quad [\varphi]_k^0 \equiv \varphi_k \\ x_k^{[1]} & := & [\varphi]_k^1(x_{Q_k}^{[0]}, x_{Q_k}^{[1]}); & \\ \ldots & & & \\ x_k^{[t]} & := & [\varphi]_k^t(x_{Q_k}^{[0]}, x_{Q_k}^{[1]}, \ldots, x_{Q_k}^{[t]}); & \end{array} \tag{47}$$

où l'ensemble Q_k contient P_k et éventuellement d'autres indices (voir Table 6). On note p_i^s la variable duale associée à $x_i^{[s]}$. D'après le programme (46) ci-dessus, la phase régressive s'initialise en prenant tous les $p_i^s = 0$ sauf

$$p_{N-m+j}^t := v_j, \quad 1 \leq j \leq m.$$

La partie de la phase régressive correspondant au traitement ligne par ligne de (47) s'écrit:

$$\left[\begin{array}{l} \textbf{for} \quad s := t \quad \textbf{down to} \quad 0 \quad \textbf{do} \\ \qquad p_i^r := p_i^r + \frac{\partial [\varphi]_k^s}{\partial x_i^{[r]}} p_k^s, \quad i \in Q_k, \quad 0 \leq r \leq s. \end{array} \right. \tag{48}$$

On peut obtenir des formules plus compactes en faisant un traitement de (47) "colonne par colonne" de manière à épuiser en une fois toutes les contributions de $[\varphi]_k^s$ $(0 \leq s \leq t)$ à p_i^r dans (48). Remarquons d'abord que lorsque p_k^s est utilisé dans (48), il a sa valeur définitive. Ces valeurs de p_k^s $(0 \leq s \leq t)$ peuvent être déterminées dans un premier temps, de façon régressive, par les formules suivantes:

$$p_k^s := p_k^s + \sum_{r=s+1}^{t} \frac{\partial [\varphi]_k^r}{\partial x_k^{[s]}} p_k^r, \quad \text{pour} \quad s = t-1, \ldots, 0. \tag{49}$$

On constate en effet que $[\varphi]_k^r$ ne dépend pas de $x_k^{[s]}$, si $s \geq r$. En particulier, aucun $[\varphi]_k^r$ $(0 \leq r \leq t)$ ne dépend de $x_k^{[t]}$ et donc p_k^t n'est pas modifié par (48). On détermine ensuite les p_i^r dans un ordre quelconque par:

$$p_i^r := p_i^r + \sum_{s=r}^{t} \frac{\partial [\varphi]_k^s}{\partial x_i^{[r]}} p_k^s, \quad \text{pour} \quad i \in Q_k, \quad 0 \leq r \leq t. \tag{50}$$

On pourra utiliser le même principe pour dériver les formules des variables duales lorsque les termes du développement de plusieurs variables doivent être évalués simultanément, comme c'est le cas pour le sinus et le cosinus. L'expression de ces formules pour quelques opérations courantes est donnée en annexe B (Table 7).

Le temps de calcul pour l'évaluation des formules (49) et (50) peut être estimé pour t grand à au plus deux fois celui de l'évaluation directe de $[\varphi_k]$, lorsque φ_k est une opération usuelle. Cette estimation est vérifiée pour les formules des tables 6 et 7: le nombre d'opérations est au plus de l'ordre de $t^2/2$ dans la table 6 et au plus de l'ordre de t^2 dans la table 7. En ce qui concerne l'espace mémoire, il faudra, comme pour le mode inverse standard $(t = 1)$, mémoriser les opérations effectuées pendant la phase progressive. Il faut aussi mémoriser les variables primales $x_i^{[s]}$ et les variables duales p_i^s, $1 \leq i \leq N$, $0 \leq s \leq t$. Au total, l'espace mémore sera en $\mathcal{O}(E) + \mathcal{O}(Nt)$, où E est le nombre d'instructions élémentaires exécutées.

2.4.3 Processus itératifs

Le contenu de cette section a été publié dans [18].

Nous entendons par *processus itératif*, une partie de programme dont les instructions sont exécutées plusieurs fois jusqu'à ce qu'un *critère d'arrêt* soit satisfait. Celui-ci peut par exemple porter sur des variables intermédiaires non spécifiées. Aussi, le nombre d'itérations n'est pas supposé être connu avant l'exécution du programme, mais sera typiquement fonction de la valeur donnée aux variables indépendantes.

Désignons par $f_{(k)} : I\!R^n \to I\!R^m$ la fonction réalisée par le code lorsque k itérations sont effectuées (conceptuellement, on peut supposer que le code ne contient qu'un seul processus itératif). Nous allons nous intéresser au cas où le processus itératif vise à ce que $f_{(k)}$ approche au mieux une certaine fonction $f : I\!R^n \to I\!R^m$. Si l'algorithne est bien conçu, on peut s'attendre à ce que, dans les circonstances normales, on ait localement la *convergence simple* de la suite $\{f_{(k)}\}$ vers f, c'est-à-dire:

$$\forall x \in \Omega, f_{(k)}(x) \to f(x), \text{ lorsque } k \to \infty, \tag{51}$$

où Ω est un ouvert de $I\!R^n$. Si un différentiateur automatique est utilisé sur un tel programme, il associera à $f_{(k)}(x)$, sa dérivée $f'_{(k)}(x)$. On peut alors se demander dans quelle mesure le

fait que $f_{(k)}(x)$ soit proche de $f(x)$ implique que $f'_{(k)}(x)$ sera proche de $f'(x)$. En d'autres termes, peut-on s'attendre à ce qu'une convergence assez forte de $f_{(k)}$ vers f implique, par exemple, la convergence simple de $f'_{(k)}$ vers f':

$$\forall x \in \Omega, f'_{(k)}(x) \to f'(x), \text{ lorsque } k \to \infty. \tag{52}$$

En toute généralité, il n'y a aucune raison pour que (51) implique (52). Pire, même si $f'_{(k)}$ converge simplement, ce peut être vers une fonction différente de f'. En fait, il faudrait avoir la convergence uniforme locale de $f'_{(k)}$ pour éviter cette situation – voir par exemple Schwartz [62, Chapitre IV, Théorème 111]. Pour illustrer cela, prenons l'exemple suivant[2] où $n = m = 1$ et

$$f_{(k)}(x) = xe^{-kx^2}. \tag{53}$$

Les fonctions $f_{(k)}$ convergent *uniformément* vers la fonction $f \equiv 0$, $f'_{(k)}(x) \to 0$ si $x \neq 0$, mais $f'_{(k)}(0) = 1$ pour tout indice k. On voit donc que la limite simple des $f'_{(k)}$ diffère de la fonction $f' \equiv 0$. Notons que lorsque les composantes des fonctions $f_{(k)}$ sont régulières et *convexes*, la propriété (51) implique la propriété (52) – voir Rockafellar [60, Théorème 24.5].

La proposition suivante montre toutefois que la situation n'est pas si fâcheuse que cela et que, pour une grande classe de processus itératifs convergents, la dérivée calculée par différentiation automatique sera asymptotiquement correcte.

Afin d'être plus précis, il nous faut quelque peu reformuler le problème en concentrant notre attention sur la partie du code représentant le processus itératif. Partageons les variables existant après la k-ième itération en un triplet $(y_{(k)}, a_{(k)}, v)$, où le vecteur v reprend les variables qui ne sont pas modifiées au cours des itérations, alors que les variables formant les vecteurs $y_{(k)}$ et $a_{(k)}$ sont mises à jour à chaque itération. Parmi ces dernières, on a fait une distinction entre les variables auxiliaires $a_{(k)}$ dont on peut se passer dans la définition de $f_{(k)}$ et celles $y_{(k)}$ dont on ne peut ou ne veut pas se passer. En d'autres termes, on doit pouvoir définir $f_{(k)}(x)$ comme fonction de $y_{(k)}$ et de v (mais pas de $a_{(k)}$):

$$f_{(k)}(x) = \psi(y_{(k)}, v), \tag{54}$$

la dépendance de $f_{(k)}$ en x venant de celle de $y_{(k)}$ et de v. On verra plus loin l'intérêt de pouvoir exprimer $f_{(k)}(x)$ indépendamment de certaines variables auxiliaires $a_{(k)}$, *en ce qui concerne l'analyse de la convergence* de $f'_{(k)}(x)$ vers $f'(x)$. Bien sûr cette structuration du code ne doit pas être connue du différentiateur automatique dont le rôle est toujours de différentier "aveuglément".

Supposons ensuite que l'on puisse décrire la mise à jour de $y_{(k)} \in I\!R^q$ au cours de l'itération k par un opérateur Φ défini sur un ouvert de $I\!R^q \times I\!R^p$ à valeurs dans $I\!R^q$:

$$y_{(k+1)} = \Phi(y_{(k)}, u), \quad k \geq 0.$$

Ici, les variables formant le vecteur $u \in I\!R^p$ sont supposées faire partie des variables formant le vecteur v et ne sont donc pas modifiées au cours des itérations. On supposera également que le point initial $y_{(0)}$ ne dépend que de u. Alors, par récurrence, $y_{(k)}$ $(k \geq 1)$ ne dépendra que de u et on peut récrire le processus itératif comme suit:

$$y_{(k+1)}(u) = \Phi(y_{(k)}(u), u), \quad k \geq 0. \tag{55}$$

[2] Cet exemple a été donné par H. Fischer au "1991 SIAM Worshop on Automatic Differentiation of Algorithms: Theory, Implementation and Application", Breckenridge, Colorado.

Si ce processus converge, c'est-à-dire si la suite $\{y_{(k)}(u)\}_{k\geq 0}$ de $\mathbb{R}^q$ converge vers un vecteur $\bar{y} = \bar{y}(u)$, celui-ci sera nécessairement un *point fixe* de $\Phi(\cdot, u)$:

$$\bar{y}(u) = \Phi(\bar{y}(u), u). \tag{56}$$

L'application $u \to \bar{y}(u)$ apparaît donc comme une fonction implicite de l'équation $y = \Phi(y, u)$.

D'après (54), on obtient pour x fixé et k tendant vers l'infini:

$$f_{(k)}(x) \to f(x) := \psi(\bar{y}(u), v).$$

Il s'agit de voir dans quelle mesure $f'_{(k)}(x)$ converge vers $f'(x)$. Si on désigne par ψ'_y et ψ'_v les dérivées partielles de ψ, on a

$$f'_{(k)}(x) = \psi'_y(y_{(k)}(u), v) \cdot y'_{(k)}(u) \cdot u'(x) + \psi'_v(y_{(k)}(u), v) \cdot v'(x),$$

$$f'(x) = \psi'_y(\bar{y}(u), v) \cdot \bar{y}'(u) \cdot u'(x) + \psi'_v(\bar{y}(u), v) \cdot v'(x).$$

On voit donc que la convergence des dérivées $f'_{(k)} \to f'$ sera assurée lorsque $y'_{(k)}(u) \to \bar{y}'(u)$. La proposition suivante donne des conditions sur Φ pour que l'on ait $y'_{(k)}(u) \to \bar{y}'(u)$.

Proposition 2.5 *Supposons que l'application Φ définie sur le produit d'ouverts $\omega_y \times \omega_u \subset \mathbb{R}^q \times \mathbb{R}^p$ à valeurs dans $\mathbb{R}^q$ soit continûment différentiable et que son application dérivée $\Phi' : \omega_y \times \omega_u \to \mathcal{L}(\mathbb{R}^q \times \mathbb{R}^p, \mathbb{R}^q)$ soit lipschitzienne. On suppose également que l'application donnant l'itéré initial, $u \to y_{(0)}(u)$, est différentiable sur ω_u et que, pour $u \in \omega_u$, la suite $\{y_{(k)}(u)\}_{k\geq 0}$ définie par (55) est dans ω_y et converge vers un point $\bar{y} = \bar{y}(u) \in \omega_y$. Si le rayon spectral ρ de la restriction $\Phi'_y(\bar{y}, u)$ de $\Phi'(\bar{y}, u)$ à $\mathbb{R}^q$ (dérivée partielle par rapport à y) vérifie*

$$\rho\left(\Phi'_y(\bar{y}, u)\right) < r < 1, \tag{57}$$

alors

(i) la convergence de la suite $\{y_{(k)}\}_{k\geq 0}$ est asymptotiquement linéaire, i.e., il existe une norme matricielle subordonnée $\|\cdot\|$ et un indice k_0 tel que $\|y_{(k+1)} - \bar{y}\| \leq r\|y_{(k)} - \bar{y}\|$, pour tout indice $k \geq k_0$,

(ii) la suite des dérivées $\{y'_{(k)}(u)\}$ converge vers $\bar{y}'(u)$.

Preuve. Montrons d'abord que la convergence de $y_{(k)}$ est asymptotiquement linéaire. D'après la formule de Taylor, on a

$$y_{(k+1)} - \bar{y} = \Phi(y_{(k)}, u) - \Phi(\bar{y}, u) = \int_0^1 \Phi'_y(\bar{y} + t(y_{(k)} - \bar{y}), u) \cdot (y_{(k)} - \bar{y})\, dt.$$

D'après (57), on peut trouver une norme matricielle subordonnée $\|\cdot\|$ telle que $\|\Phi'_y(\bar{y}, u)\| < r < 1$. Alors, dès que $y_{(k)}$ est assez proche de $\bar{y}$, disons pour $k \geq k_0$, on a

$$\|\Phi'_y(\bar{y} + t(y_{(k)} - \bar{y}), u)\| \leq r, \quad \text{pour } t \in [0, 1],$$

et donc pour une norme vectorielle convenable:

$$\|y_{(k+1)} - \bar{y}\| \leq r\, \|y_{(k)} - \bar{y}\|, \quad \text{pour } k \geq k_0,$$

ce qui montre (i).

Avant de prouver (ii), remarquons que l'application $u \to y_{(k)}(u)$ est différentiable comme composée de k applications différentiables avec l'application différentiable $u \to y_{(0)}(u)$. L'application $u \to \bar{y}(u)$ est également différentiable car il s'agit d'une fonction implicite de l'équation $F(y,u) \equiv y - \Phi(y,u) = 0$ et que l'on est dans les conditions d'application du théorème des fonctions implicites. En effet, les conditions de régularité étant remplies, il reste à vérifier que $F'_y(\bar{y},u) = I - \Phi'_y(\bar{y},u)$ est inversible. Il suffit de vérifier l'injectivité de cet opérateur: si $\eta \in I\!R^q$ vérifie $F'_y(\bar{y},u)\cdot\eta = 0$, alors $\eta = \Phi'_y(\bar{y},u)\cdot\eta$ et, en utilisant (57), on a $\|\eta\| \le r\|\eta\|$, ce qui implique que $\eta = 0$ (puisque $r < 1$).

On peut alors différentier (55) et (56) par rapport à u dans une direction $v \in I\!R^p$. En notant $\xi = \bar{y}'(u)\cdot v$ et $\xi_{(k)} = y'_{(k)}(u)\cdot v$, on a

$$\xi_{(k+1)} = \Phi'_y(y_{(k)},u)\cdot\xi_{(k)} + \Phi'_u(y_{(k)},u)\cdot v,$$

$$\xi = \Phi'_y(\bar{y},u)\cdot\xi + \Phi'_u(\bar{y},u)\cdot v.$$

En retranchant ces deux identités membre à membre, on obtient:

$$\xi_{(k+1)} - \xi = \Phi'_y(y_{(k)},u)\cdot[\xi_{(k)} - \xi] + [\Phi'_y(y_{(k)},u) - \Phi'_y(\bar{y},u)]\cdot\xi + [\Phi'_u(y_{(k)},u) - \Phi'_u(\bar{y},u)]\cdot v.$$

Grâce au caractère lipschitzien de Φ' et à (57), on obtient pour k assez grand et une constante positive C:

$$\|\xi_{(k+1)} - \xi\| \le r\|\xi_{(k)} - \xi\| + C\|y_{(k)} - \bar{y}\|.$$

Par récurrence et en utilisant (i), on trouve pour $k \ge k_1 \ge k_0$:

$$\begin{aligned}\|\xi_{(k+1)} - \xi\| &\le r^{k-k_1+1}\|\xi_{(k_1)} - \xi\| + C\left(\sum_{i=k_1}^{k} r^{k-i}\|y_{(i)} - \bar{y}\|\right)\\ &\le r^{k-k_1+1}\|\xi_{(k_1)} - \xi\| + C\,(k-k_1+1)r^{k-k_1}\|y_{(k_1)} - \bar{y}\|.\end{aligned}$$

Comme $r < 1$, on a $r^{k-k_1+1} \to 0$ et $(k-k_1+1)r^{k-k_1} \to 0$ lorsque k tend vers l'infini, ce qui montre que $\xi_{(k)} = y'_{(k)}(u)\cdot v$ converge vers $\xi = \bar{y}'(u)\cdot v$. Comme ceci est vrai pour tout $v \in I\!R^p$, on en déduit que (on est en dimension finie) $y'_{(k)}(u) \to \bar{y}'(u)$ dans $\mathcal{L}(I\!R^p, I\!R^q)$. □

Un résultat similaire a été donné par Fischer [14], lorsque le processus itératif vise à résoudre un système linéaire.

Remarquons qu'on ne peut pas se passer d'une hypothèse du type (57). En effet, les itérations (53) fournissent un contre-exemple limite pour cette hypothèse, puisque celles-ci s'écrivent $f_{(k+1)} = \Phi(f_{(k)},x)$, avec

$$\Phi(f,x) = fe^{-x^2}, \quad f_{(0)}(x) = x.$$

On voit que $\Phi'_f(f,0) = e^{-x^2}\Big|_{x=0} = 1$ ne vérifie pas l'hypothèse (57).

Comme exemple d'application de la proposition, considérons le cas où l'on utilise des itérations de Newton pour calculer une fonction implicite, solution $y : u \in I\!R^p \to y(u) \in I\!R^q$ de

$$F(y,u) = 0. \tag{58}$$

Supposons que l'application $F : \omega_y \times \omega_u \to \mathbb{R}^q$ soit suffisamment régulière et que le point $y_{(0)}$ soit obtenu comme fonction régulière de u. Les itérés de Newton se calculent au moyen de la formule

$$y_{(k+1)} = y_{(k)} - F'_y(y_{(k)}, u)^{-1} F(y_{(k)}, u). \tag{59}$$

On sait que si le point $y_{(0)}(u)$, $u \in \omega_u$, est suffisamment proche d'un point $\bar{y}(u)$ vérifiant $F(\bar{y}(u), u) = 0$, la suite $\{y_{(k)}(u)\}_{k \geq 0}$ convergera vers $\bar{y}(u)$. Ici, la fonction Φ de la proposition s'écrit

$$\Phi(y, u) = y - F'_y(y, u)^{-1} F(y, u).$$

Elle vérifie la condition (57) puisque, grâce à $F(\bar{y}, u) = 0$, on a

$$\Phi'_y(\bar{y}, u) = I + F'_y(\bar{y}, u)^{-1} F''_{yy}(\bar{y}, u) F'_y(\bar{y}, u)^{-1} F(\bar{y}, u) - F'_y(\bar{y}, u)^{-1} F'_y(\bar{y}, u) = 0.$$

Par conséquent, la suite $\{y'_{(k)}(u)\}_{k \geq 0}$ convergera vers $\bar{y}'(u)$.

Ce dernier exemple montre l'intérêt de faire une distinction entre variables mises à jour $y_{(k)}$ dont dépend $f_{(k)}$ et celles $a_{(k)}$ dont $f_{(k)}$ ne dépend pas directement. En effet, une mise en oeuvre des itérations de Newton (59) dans un code de calcul met en jeu beaucoup d'autres variables auxiliaires $a_{(k)}$ en plus des variables $y_{(k)}$ définies par (59). Or, d'une part, il est souvent difficile de donner une loi de mise à jour $\tilde{\Phi}$ de ces $a_{(k)}$ et d'autre part rien ne dit que la loi globale $(\Phi, \tilde{\Phi})$ vérifiera la condition (57) et cela même si cette condition est vérifiée par Φ: il suffit par exemple d'ajouter au processus des itérations (inutiles) du type (53). Or, comme nous l'avons vu, une telle condition sur $(\Phi, \tilde{\Phi})$ est inutile si $f_{(k)}$ ne dépend pas de $a_{(k)}$.

Nous avons abordé ci-dessus la question de l'exactitude des dérivées calculées par différentiation automatique en présence d'un processus itératif. Il nous reste à discuter de l'efficacité de la différentiation automatique dans de telles circonstances. A première vue, il semble qu'il faille mener le processus de différentiation tout au long des itérations ayant conduit au résultat. Ceci serait très coûteux. Cependant, comme l'a fait remarquer Griewank [24], il est des cas où il suffit de relancer le processus itératif pendant quelques itérations, et de différentier le programme exécutant ces quelques itérations. Considérons, en effet, le processus de Newton (59). Si celui-ci a été interrompu à l'itération k avec $y_{(k)} \approx \bar{y}$, il suffit de différentier le programme faisant une *unique* itération de Newton à partir de $y_{(k)}$:

$$z = y_{(k)} - F'_y(y_{(k)}, u)^{-1} F(y_{(k)}, u).$$

Dans ce programme $y_{(k)}$ est considéré comme constant et comme $F(y_{(k)}, u) \approx F(\bar{y}, u) = 0$, la dérivée de z en u s'écrit:

$$\begin{aligned} z'(u) \cdot v &= -F'_y(y_{(k)}, u)^{-1} \left(F''_{yu}(y_{(k)}, u) \cdot v \right) F'_y(y_{(k)}, u)^{-1} F(y_{(k)}, u) \\ &\quad -F'_y(y_{(k)}, u)^{-1} F'_u(y_{(k)}, u) \cdot v \\ &\approx -F'_y(\bar{y}, u)^{-1} F'_u(\bar{y}, u) \cdot v, \end{aligned}$$

qui est bien la dérivée $\bar{y}'(u) \cdot v$ cherchée.

3 Analyse de la sensibilité aux erreurs d'arrondi

Comme nous l'avons mentionné précédemment (Section 2.3), le mode inverse de la différentiation automatique fournit des quantités permettant une analyse de la propagation des erreurs

d'arrondi dans un programme de calcul numérique. Nous nous proposons dans cette section d'expliciter les quantités en question. Nous commençons par un rapide survol des travaux menés pour l'analyse des erreurs d'arrondi, ayant abouti à des réalisations logicielles: il ne s'agit pas d'une revue exhaustive mais plutôt d'un recensement partiel centré sur les aspects pratiques. Nous donnerons ensuite les résultats principaux permettant de mettre en lumière les apports de la différentiation automatique au problème de l'analyse d'erreur d'arrondi. Rappelons enfin que [72] reste une référence essentielle en matière d'analyse d'erreur, tant d'un point de vue théorique que pratique.

3.1 Les études sur l'analyse des erreurs d'arrondi

Les études menées depuis les années 70 sur l'analyse de la génération et de la propagation des erreurs d'arrondi dans les programmes numériques se sont faites essentiellement autour de la notion de graphe de calcul ([1]). Cependant, la complexité du problème, en raison de la nécessité du calcul des dérivées partielles des fonctions par rapport à toutes les variables intermédiaires intervenant au cours du déroulement du programme, n'a pas permis de mise en pratique efficace des résultats de ces études. En effet, pour des fonctions quelque peu complexes, le volume de calcul nécessaire pour l'évaluation de ces dérivées (par dérivation symbolique ou numérique) rend les méthodes issues de ces études rapidement impraticables. Quelques tentatives ont été faites dans ce sens, notamment autour de J. Larson, A. Sameh ([41], [43], [44], [42]) et W. Miller ([49], [50]), où des outils logiciels ont été développés pour permettre une analyse des erreurs d'arrondi dans des programmes FORTRAN. Les logiciels produits pour de telles analyses ont pour inconvénients majeurs qu'ils n'analysent que des programmes écrits dans un pseudo-langage (SPL, "voisin" de FORTRAN dans le cas de la méthode de Miller) – qu'il faut donc apprendre ! – et qu'ils ne permettent d'étudier que les algorithmes *straight-line*. Les codes qu'ils permettent d'analyser ne peuvent dépasser la centaine de lignes d'instruction et ne peuvent comporter qu'un sous-ensemble réduit du FORTRAN (pas d'instructions de branchement par exemple): on conçoit aisément que de tels outils, d'usage difficile (on doit par exemple fournir le graphe de calcul en entrée du logiciel de Miller) et lents à l'exécution, ne présentent qu'un intérêt limité lorsque l'on cherche à analyser des programmes généraux. Une analyse du logiciel de Miller et des résultats obtenus a été faite par Bonnet (1989, 1990), travaux auxquels nous renvoyons pour plus de détails. L'idée générale est de simuler, à partir du graphe, l'effet des erreurs d'arrondi dues aux opérations arithmétiques, par perturbation des données et/ou des résultats. Pour résumer, l'idée ancienne est ainsi reprise de montrer que le résultat calculé est le résultat exact obtenu avec des données perturbées (*cf.* [20] e.g.).

Plus récemment des développements logiciels ont été menés ([3]) pour générer le graphe de calcul à partir d'un programme FORTRAN et permettre par la suite une utilisation du logiciel de Larson-Sameh pour l'analyse des erreurs d'arrondi mais aussi pour le comptage des opérations dans un programme, l'analyse par intervalles. Il s'agit du développement d'un préprocesseur spécifique à une machine (Alliant FX/8) qui utilise les particularités du compilateur vectoriel de cette machine.

Hormis les travaux menés autour de Miller d'une part, Larson-Sameh d'autre part, des études ont été menées autour de Iri pour l'analyse des erreurs d'arrondi au moyen des techniques de différentiation automatique. Ces travaux ont abouti en particulier à la réalisation d'un précompilateur (PADRE2) dont nous reparlerons plus loin à propos des réalisations ef-

fectives (Section 5.3). On pourra également consulter les travaux de Vignes (point d'entrée dans [69]).

3.2 Analyse d'erreur et différentiation automatique

Nous abordons maintenant l'analyse des erreurs d'arrondi telle que développée autour de Iri essentiellement (*cf.* [30], [28], [33] et [32]). Nous suivrons en particulier les lignes de l'étude faite par Hoshi, Iri et Tsuchiya [28]. Pour simplifier l'exposé, nous supposerons que le programme calculant la fonction f obéit au modèle simple (Section 2.1.1).

Notations: Outre les notations introduites dans les sections précédentes, nous noterons dans cette section par $\hat{x}$ la valeur effectivement calculée d'une variable x, résultat d'une opération en précision finie, et la fonction intermédiaire donnant cette valeur sera notée $\hat{\varphi}$. Les instructions effectivement calculées dans le code s'écrivent donc $\hat{x}_k := \hat{\varphi}_k(\hat{x}_{P_k})$. L'erreur commise sur une variable x sera notée $\tilde{x}$ (i.e., $\tilde{x} = \hat{x} - x$).

Nous pouvons maintenant, compte tenu de ce qui précède, associer à chaque variable (indépendante ou intermédiaire), sa valeur effectivement calculée ainsi que l'erreur commise lors du calcul:

$$\begin{aligned} \tilde{x}_k &= \hat{x}_k - x_k \\ &= \hat{\varphi}_k(\hat{x}_{P_k}) - \varphi_k(x_{P_k}) \\ &= [\varphi_k(\hat{x}_{P_k}) - \varphi_k(x_{P_k})] + \delta x_k, \end{aligned} \tag{60}$$

où,

$$\delta x_k = \hat{\varphi}_k(\hat{x}_{P_k}) - \varphi_k(\hat{x}_{P_k}).$$

En supposant, ce qui est raisonnable, que $\tilde{x}_j$ est petit pour $j \in P_k$, on peut développer l'expression (60) au premier ordre et obtenir ainsi:

$$\tilde{x}_k \simeq \sum_{j \in P_k} \frac{\partial \varphi_k}{\partial x_j}(x_{P_k})\,(\hat{x}_j - x_j) + \delta x_k = \sum_{j \in P_k} \frac{\partial \varphi_k}{\partial x_j}\,\tilde{x}_j + \delta x_k. \tag{61}$$

Suivant [28], de façon naturelle, le premier terme du membre de droite de (61) est appelé l'*erreur propagée* jusqu'à l'étape k et le second terme l'*erreur générée* à l'étape k. Ayant ainsi obtenu une expression de l'erreur à une étape quelconque, on est désormais en mesure de donner l'expression de l'*erreur globale* commise au cours du calcul de la fonction f (dont on a supposé qu'elle était scalaire, l'extension au cas d'une fonction à valeurs vectorielles étant immédiate par application des calculs précédents à chacune des composantes de f), par application répétée de la formule (61). Si on suppose que $\tilde{x}_i = 0$, pour $1 \leq i \leq n$ (on exclut ainsi les variables indépendantes ou variables d'entrée du problème, parce que l'on ne s'intéresse qu'aux erreurs générées par le programme de calcul), on trouve:

$$\tilde{f} \simeq \sum_{k=n+1}^{N} \left[\sum_{\substack{\text{chemin } \mathcal{C} \\ \text{de } x_k \text{ à } x_N}} \prod_{\text{arc } (i,j) \in \mathcal{C}} \frac{\partial \varphi_i}{\partial x_j} \right] \delta x_k.$$

Compte tenu de (8), on obtient:

$$\tilde{f} \simeq \sum_{k=n+1}^{N} \frac{\partial f}{\partial x_k} \delta x_k. \tag{62}$$

où la somme est effectuée sur toutes les variables dépendantes.

Nous voyons ainsi apparaître dans les calculs précédents les quantités $p_k =$ "$\partial f / \partial x_k$", $k = n+1, \ldots, N$, "dérivées" de la fonction f par rapport à toutes les variables intermédiaires x_k. Or nous avons vu (Section 2.3) que ces quantités étaient accessibles par les méthodes de différentiation automatique en mode inverse: ce sont les quantités p_k obtenues dans l'algorithme DAI1. Le mode inverse de la différentiation automatique fournit donc une partie des quantités permettant une évaluation de $\tilde{f}$, l'erreur commise sur f.

Pour mener à bien le calcul d'estimateurs d'erreur, quelques hypothèses classiques sont généralement faites sur les erreurs générées lors d'une étape de calcul. Nous les résumons ci-dessous et donnons un exemple de calcul d'estimateur, à partir de ces hypothèses. On trouvera en [28] et [33] des analyses détaillées et les calculs correspondants d'estimateurs probabilistes des erreurs pour l'arrondi et la troncature.

Hypothèses

- Les calculs étant faits en arithmétique flottante, un réel admet une représentation avec mantisse et exposant:

  $$x_i = \mathrm{sgn}(x_i) M b^e \tag{63}$$

  où e est l'exposant (entier), M la mantisse et b la base (2,10 ou 16), normalisée de sorte que $b^{-1} \leq M < 1$.

- Le résultat d'une "opération de base" est arrondi au nombre "le plus proche" (arrondi ou troncature), représentable sous la forme précédente.

- L'erreur commise sur le résultat d'un calcul est supposée être la réalisation d'une variable aléatoire de loi uniforme (sur un intervalle dépendant du sens que l'on donne à "le plus proche") et les erreurs commises à chaque pas de calcul sont indépendantes entre elles.

Compte tenu de ces trois hypothèses, les calculs donnés en [28] fournissent deux résultats d'estimation de l'erreur sur le calcul de f: une borne sur la valeur absolue de l'erreur et une estimation des deux premiers moments de cette erreur, considérée comme variable aléatoire. Donnons par exemple le calcul d'une estimation de la moyenne et de la variance de l'erreur commise sur le calcul de f; de l'expression (62), on déduit:

$$\begin{aligned} E(\tilde{f}) &\simeq \sum_{k=n+1}^{N} \frac{\partial f}{\partial x_k} E(\delta x_k) \\ V(\tilde{f}) &\simeq \sum_{k=n+1}^{N} \frac{\partial f}{\partial x_k} V(\delta x_k) \end{aligned}$$

Dans l'hypothèse d'une distribution uniforme sur $[-\epsilon_k, +\epsilon_k]$ pour δx_k, on a bien sûr $E(\delta x_k) = 0$ et donc $E(\tilde{f}) = 0$; par ailleurs on a: $V(\delta x_k) = \epsilon_k^2/3$ et donc:

$$\begin{aligned} E(\tilde{f}^2) &= V(\tilde{f}) \\ &= \frac{1}{3}\epsilon^2 \sum_{k=n+1}^{N} \left|\frac{\partial f}{\partial x_k}\right|^2 m(x_k)^2 \\ &\leq \frac{1}{3}\epsilon^2 \sum_{k=n+1}^{N} \left|\frac{\partial f}{\partial x_k}\right|^2 |x_k|^2 \end{aligned}$$

où $m(x)$ est le plus petit nombre non négatif (dans la représentation (63)) ayant le même exposant que x. On a en particulier: $m(x) \leq |x| < b.m(x)$ où b est la base de la représentation des nombres en flottant, définie en (63). On trouvera en [28] des calculs analogues pour le calcul d'une "borne absolue de l'erreur" ou lorsque le support de la distribution de probabilité pour δx_i n'est plus $[-\epsilon_i, +\epsilon_i]$ mais $[0, \epsilon_i]$ ou $[-\epsilon_i, 0]$ par exemple.

De façon pratique, c'est ce type de résultat que le différentiateur automatique PADRE2 fournit, comme nous le verrons plus loin sur les cas-tests.

En conclusion de cette section, on peut dire que la différentiation automatique en mode inverse, outre son intérêt pour le calcul de dérivées qui en est la motivation première, fournit de façon efficace (en ce qui concerne le temps d'exécution, voir Section 2.3.9) les éléments nécessaires pour une analyse des erreurs survenant au cours du déroulement d'un programme.

4 Techniques d'implémentation

4.1 La précompilation

L'utilisation d'un précompilateur pour la différentiation automatique de fonctions représentées par des programmes se décompose typiquement en trois étapes:

étape 1: adaptation éventuelle du précompilateur;
étape 2: précompilation de la partie du programme qui calcule la fonction;
étape 3: modification éventuelle du code appelant la partie du programme ayant été précompilée et compilation du tout avec des *sous-routines d'accompagnement* (ou "runtime").

Nous allons décrire brièvement la manière dont on peut implémenter le mode inverse par précompilation. Le mode direct, plus simple, pourra l'être en suivant une démarche analogue. La difficulté de mise en oeuvre du mode inverse vient du fait qu'il faut mémoriser des informations au moment du calcul de la fonction, celles-ci devant être utilisées plus tard, au cours de la phase régressive qui débute à l'achèvement du calcul de la fonction. Ce délai imposé donne lieu à plusieurs méthodes. En gros, on en distingue deux.

4.1.1 La méthode du graphe de calcul

C'est la méthode la plus simple à implémenter. On mémorise *séquentiellement* toutes les quantités utiles à la phase régressive dans une base de données structurée. Celle-ci peut-être vue comme une représentation du graphe de calcul de la fonction. La phase régressive

est alors un algorithme assez simple, prenant la forme d'une boucle répétitive (comme dans l'algorithme DAI1) qui exploite en sens inverse le contenu informatif de la structure ainsi créée.

Par exemple, dans le précompilateur JAKEF (section 5.1), toutes les instructions sont décomposées en instructions élémentaires φ_k ayant 0, 1 ou 2 arguments et on mémorise dans une pile le nombre d'arguments de chaque φ_k, les variables x_j ($j \in P_k$) dont φ_k dépend et les dérivées partielles $\partial\varphi_k/\partial x_j$. Dans le précompilateur PADRE2 (section 5.3), c'est le type de l'opération effectuée par φ_k qui est mémorisée, les dérivées partielles $\partial\varphi_k/\partial x_j$ pouvant s'en déduire.

Avec cette méthode, la place mémoire requise est proportionnelle au nombre d'instructions élémentaires exécutées et pourra donc être très importante, même dans des cas simples (voir l'exemple de la section 6.3).

4.1.2 La méthode du code adjoint

La méthode de précompilation que nous présentons sous cette dénomination consiste à mettre en oeuvre, de la façon la plus immédiate, la différentiation automatique en mode inverse introduite en section 2.3 et développée en section 4.3. Partant d'un code calculant une fonction ("*code primal*"), on écrit un *code adjoint*, en "dualisant" ligne par ligne les instructions du code primal, dans l'ordre *inverse* de leur *exécution*. La phase régressive n'est donc plus une unique boucle répétive (comme celle présentée dans l'algorithme DAI2) travaillant sur une grande base de données, mais devient une longue suite d'instructions *structurées*, travaillant (comme nous allons le voir) sur un nombre beaucoup plus restreint de données. En passant de la méthode du graphe de calcul à la méthode du code adjoint, on a, suivant un principe bien connu, échangé de l'information contre de l'algorithmique.

Nous verrons en section 4.3 qu'avec cette méthode on retrouve dans le code adjoint une partie de la structure du code primal. Par exemple, une boucle répétitive deviendra une autre boucle répétitive dont l'indice de boucle variera en sens inverse de celui de la boucle primale. De même, une instruction conditionnelle `if-then-else` deviendra une autre instruction conditionnelle pourtant sur le même test.

A notre connaissance, cette technique a été introduite par Ph. Courtier et O. Talagrand. Certains météorologistes l'utilisent depuis plusieurs années sur des codes écrits en FORTRAN – *cf.* Talagrand (1991) qui la décrit sur des exemples instructifs. Malheureusement, l'écriture du code adjoint se fait encore manuellement alors que, à part quelques choix que nous allons préciser, celle-ci ne demande au programmeur que de suivre aveuglément des règles non ambiguës. Il semble qu'il n'y ait pas actuellement (fin 1991) d'implémentation automatique de cette technique, mais une percée encourageante a été réalisée par S. Dalmas et N. Rostaing ([10]).

L'avantage majeur de la méthode du code adjoint est sa sobriété en place mémoire. Curieusement, ceci ne semble pas avoir été remarqué par les premiers concepteurs de différentiateurs automatiques. Un autre avantage est que l'absence de base de données structurée permet d'éviter les opérations de recherche et de manipulation des objets de la base, qui dans la méthode précédente donne lieu à un surcoût en temps de calcul qui peut être important.

Il y a plusieurs raisons pour lesquelles la méthode du code adjoint permet d'économiser de la place mémoire. La première vient de l'observation suivante. Pour cela rappelons les instructions à écrire dans le code adjoint, correspondant à l'instruction d'affectation primale

$x_{\mu_k} := \varphi_k(x_{P_k})$. On écrira, *dans l'ordre*,

$$\begin{cases} p_j := p_j + \dfrac{\partial \varphi_k}{\partial x_j}\, p_{\mu_k}, \forall\, j \in P_k \backslash \{\mu_k\}; \\ p_{\mu_k} := \dfrac{\partial \varphi_k}{\partial x_{\mu_k}}\, p_{\mu_k}, \end{cases}$$

où p_j est la variable duale associée à la variable x_j. On voit que si x_j, $j \in P_k$, n'intervient que de façon *linéaire* dans φ_k, sa valeur n'intervient pas dans les instructions dualisant φ_k. En particulier, la dérivée paritelle $\partial\varphi_k/\partial x_j$ est soit une constante, soit une quantité ne dépendant pas des variables indépendantes. Dans le premier cas, il suffira d'utiliser la même constante dans le code adjoint, dans le second cas on utilisera la même quantité, si celle-ci n'est pas modifiée après exécution de l'instruction φ_k courante. Par conséquent, la méthode du code adjoint a la propriété agréable de ne devoir en général mémoriser aucune quantité calculée dans la phase progressive si la fonction à différentier est *linéaire*.

Une autre raison vient de ce qu'au lieu de mémoriser certaines variables, il est parfois préférable de les recalculer au moment où elles s'avèrent nécessaires dans la phase régressive. Pour illustrer cela, considérons le programme suivant contenant deux boucles emboîtées:

$$\left[\begin{array}{l} x_s := 0; \\ \textbf{for}\ \ i_1 := 1\ \ \textbf{to}\ \ l_1\ \ \textbf{do} \\ \quad \textbf{for}\ \ i_2 := 1\ \ \textbf{to}\ \ l_2\ \ \textbf{do}\ \ \{ \\ \qquad x_t := x_{j(i_1)}\, x_{k(i_2)}; \\ \qquad \cdots \\ \qquad x_s := x_s + x_t^2; \\ \qquad \cdots \\ \quad \} \end{array}\right.$$

et supposons que les variables $x_{j(i_1)}, 1 \le i_1 \le l_1$ et $x_{k(i_2)}, 1 \le i_2 \le l_2$ aient été mémorisées. La variable "temporaire" x_t intervient de façon non linéaire et doit donc, ou bien être mémorisée, ou bien être recalculée. Dans le premier cas, il faudrait mémoriser toutes les valeurs prises par $x_t := x_{j(i_1)}\, x_{k(i_2)}$, lorsque $i_1 = 1 \ldots l_1$ et $i_2 = 1 \ldots l_2$, ce qui est très encombrant dès que l_1 et l_2 sont grands. On voit que dans ce cas il est préférable de les recalculer. Le code adjoint du code précédent s'écrira donc, en utilisant les règles détaillées en section 4.3:

$$\left[\begin{array}{l} \textbf{for}\ \ i_1 := l_1\ \ \textbf{down to}\ \ 1\ \ \textbf{do} \\ \quad \textbf{for}\ \ i_2 := l_2\ \ \textbf{down to}\ \ 1\ \ \textbf{do}\ \ \{ \\ \qquad \cdots \\ \qquad x_t := x_{j(i_1)}\, x_{k(i_2)}; \\ \qquad p_t := p_t + 2\, x_t\, p_s; \\ \qquad \cdots \\ \qquad p_{j(i_1)} := p_{j(i_1)} + x_{k(i_2)}\, p_t; \\ \qquad p_{k(i_2)} := p_{k(i_2)} + x_{j(i_1)}\, p_t; \\ \qquad p_t := 0; \\ \quad \} \\ p_s := 0. \end{array}\right.$$

Un exemple complet avec ce type de choix est donné en section 6.3. Rappelons que la méthode du graphe de calcul ne connait pas cette souplesse: dans cet exemple, on y choisit systématiquement la mémorisation de toutes les valeurs prises par x_t.

Une dernière raison est qu'il n'est plus nécessaire de mémoriser la structure des instructions d'affectation φ_k. Dans la première méthode de précompilation, cette structure était mémorisée dans le graphe de calcul après avoir décomposé φ_k en fonctions élémentaires. Ici, la structure de φ_k est directement utilisée par le programmeur pour composer le code adjoint. La partie utile de la structure de φ_k se retrouve exprimée sous forme de programme et non pas dans une base de données.

La méthode du code adjoint a également l'intérêt de permettre la création de nouveaux algorithmes, pourvu que la fonction réalisée par ceux-ci puisse s'*interpréter* comme le gradient d'une fonction représentable par un programme ayant la structure (2). Pour une application de cette idée, voir [19].

4.2 La surcharge des opérateurs

Dans la différentiation automatique par précompilation, les formules sont analysées deux fois: une première fois par le précompilateur pour générer le code précompilé et une seconde fois par le compilateur pour analyser le code précompilé. Dans certains langages orientés objet, la technique dite de *surcharge des opérateurs* ("operator overloading") permet de faire les deux choses à la fois lors de la compilation. Par cette technique, le programmeur peut introduire de nouveaux types de variables ou d'objets ainsi que des opérations entre ces objets. Ces opérations ne sont pas seulement des appels de sous-routines propres aux objets introduits, mais peuvent être invoquées par les symboles opératoires et fonctions usuels: `+`, `-`, `*`, `/`, `sin`, `exp`, `=`, ... Dans ce dernier cas, on dit que les opérateurs sont "surchargés". Un même opérateur (resp. une même fonction) peut donc représenter différentes opérations, en fonction de ses opérandes (resp. de ses arguments). C'est précisément le type des variables en jeu qui permet au compilateur de sélectionner l'opération à effectuer.

De cette manière, on peut forcer le compilateur à générer le code nécessaire au calcul du gradient (mode direct). Pour le mode inverse, le code généré par le compilateur servira à mémoriser le graphe de calcul de la fonction. Le calcul du gradient (ou des dérivées d'ordre supérieur) se fait ensuite au moyen d'une sous-routine exploitant le graphe de calcul.

L'avantage de cette technique est de pouvoir écrire des codes très propres. La tâche de l'utilisateur est essentiellement de redéclarer les variables indépendantes et intermédiaires d'un type adéquat. Cette technique est disponible en PASCAL-SC, en FORTRAN-8X [48], en C++ [66, 12], en ADA, ...

La première utilisation de cette idée semble due à Kedem (1980), en FORTRAN. C'est le mode direct qui est implémenté. Il utilise le précompilateur AUGMENT qui permet de définir des types de données (appelés dans ce cas, TAYLOR et GRADIENT), opérateurs et fonctions non standard. Rall (1984, 1987) a développé une version plus propre de cette idée pour le mode direct, en PASCAL-SC. Voir Huss (1991) pour un début d'implémentation du mode direct en ADA.

La même idée est utilisée par Griewank et al. (1990) dans le code ADOL-C, qui permet d'utiliser les modes direct et inverse.

4.3 Traitement de quelques instructions particulières

Un code de calcul contient en général des instructions qui ne sont pas des affectations. Dans cette section, nous décrivons brièvement comment chaque type d'implémentation peut faire

face à ces instructions. Pour simplifier l'exposé, nous nous limiterons au cas du calcul de dérivées premières.

L'extension du mode direct à des programmes plus complexes que ceux obéissant au modèle étendu est aisée. Ce mode de dérivation peut prendre en compte des situations diverses: instructions d'affectation (bien sûr), branchements conditionnels, boucles répétitives, sous-routines, processus itératifs, etc .... Il suffit, on s'en convaincra, de faire *précéder* chaque instruction d'affectation de l'instruction calculant sa dérivée directionnelle (Algorithme DAD2, Section 2.2) et de maintenir telles quelles toutes autres structures du code. De cette manière, chaque fois qu'une instruction d'affectation du code original est exécutée, la dérivée directionnelle de la variable modifiée est également réévaluée.

En ce qui concerne le mode inverse, cela dépendra du type d'implémentation. Sur cette question, la méthode de l'arbre de calcul semble la plus simple. Il suffit, en effet, de construire l'arbre au cours du calcul de la fonction, en le complétant chaque fois qu'une instruction d'affectation est exécutée. De cette manière, l'arbre représentera le calcul réellement exécuté et la phase régressive correspondra au calcul du gradient de la fonction représentée par les instructions d'affectation exécutées. Des difficultés peuvent apparaître lorsque le code original contient des instructions conditionnelles et si le point d'évaluation de f n'est pas contenu dans un voisinage sur lequel le graphe de calcul est invariant. Il s'agit là d'une difficulté intrinsèque, indépendante du type d'implémentation: en de tels points la fonction peut ne pas être différentiable.

Venons-en maintenant au cas de la génération de codes adjoints. L'algorithme DAI2 met clairement en évidence le fait qu'à chaque instruction d'affectation du code primal (phase progressive) correspond un lot d'instructions "duales" dans le code adjoint (phase régressive). L'implémentation du mode inverse par code adjoint pose donc la question de la "dualisation" des instructions qui ne sont pas des affectations.

Avant d'aborder cette question délicate, il n'est pas inutile de voir comment se fait la "dualisation" de certaines instructions d'affectation. Rappelons d'abord les formules (24), qui donnent les instructions de la phase régressive correspondant à la **modification** d'une variable x_i, $i = \mu_k$, au moyen de l'instruction d'affectation

$$x_{\mu_k} := \varphi_k(x_{P_k}). \tag{64}$$

La dualisation de (64) s'écrit (on note p_j la variable duale de x_j):

$$\begin{aligned} p_j &:= p_j + \tfrac{\partial \varphi_k}{\partial x_j}\, p_{\mu_k}, \quad \forall\ j \in P_k \backslash \{\mu_k\}; \\ p_{\mu_k} &:= \tfrac{\partial \varphi_k}{\partial x_{\mu_k}}\, p_{\mu_k}, \end{aligned}$$

S'il s'agit d'une **initialisation**, $\mu_k \notin P_k$ et la dualisation s'écrira:

$$\begin{aligned} p_j &:= p_j + \tfrac{\partial \varphi_k}{\partial x_j}\, p_{\mu_k}, \quad \forall\ j \in P_k; \\ p_{\mu_k} &:= 0. \end{aligned}$$

Notons que l'instruction '$p_{\mu_k} := 0$' peut-être interprétée comme une nouvelle initialisation de la variable duale de x_{μ_k}. En fait, cette nouvelle initialisation est inutile si la variable x_{μ_k} n'est pas utilisée dans le code avant l'instruction (64) en cours de traitement, mais elle est essentielle dans le cas contraire. Enfin, le cas le plus simple est celui d'une **augmentation**,

$x_{\mu_k} := x_{\mu_k} + \tilde{\varphi}_k(x_{\tilde{P}_k})$, $\mu_k \notin \tilde{P}_k$, puisque alors, la dualisation de (64) s'écrit simplement:

$$p_j := p_j + \frac{\partial \tilde{\varphi}_k}{\partial x_j}\, p_{\mu_k}, \quad \forall\ j \in \tilde{P}_k.$$

On peut à présent considérer le cas d'un **produit scalaire**:

$x_k := 0;$
for $i := 1$ **to** l **do** $x_k := x_k + x_{r_i}\, x_{s_i},$

où l'on suppose, pour simplifier, que $k \notin \{r_i : 1 \le i \le l\} \cup \{s_i : 1 \le i \le l\}$. En utilisant les règles précédentes, on obtient dans la phase régressive:

for $i := l$ **down to** 1 **do** {
  $p_{r_i} := p_{r_i} + x_{s_i}\, p_k;$
  $p_{s_i} := p_{s_i} + x_{r_i}\, p_k;$
}
$p_k := 0.$

Ces formules restent valables même si $\{r_i : 1 \le i \le l\} \cap \{s_i : 1 \le i \le l\} \neq \emptyset$.

Plus généralement, la dualisation d'une boucle répétitive (`do`, `for`, `while`, ...) se fera par une autre boucle répétitive dans laquelle les instructions sont "dualisées" en sens inverse de leur exécution dans la boucle primale. Donc, pour la boucle `do` FORTRAN, l'indice de la boucle duale varie en sens inverse de celui utilisé dans le code original. C'est cette règle que nous avons suivie dans la dualisation du produit scalaire: les indices i décroissent de l à 1. En fait, cela est sans importance dans ce cas particulier, mais cela pourrait l'être si la valeur des variables modifiées (ici x_k) était utilisée à l'intérieur de la boucle dans des instructions qui ne seraient pas des "augmentations".

On se convaincra que la dualisation de l'instruction `if-then-else`:

if (condition C) **then** {
  instructions A
} **else** {
  instructions B
}

se fait simplement par

if (condition C) **then** {
  code adjoint de (instructions A)
} **else** {
  code adjoint de (instructions B)
}.

Le cas des instructions `goto` est délicat car le code primal peut en contenir beaucoup, si bien que l'ordre d'exécution des instructions peut paraître inextricable. Notons qu'avec des `goto`, on peut construire des processus complexes tels que les processus itératifs. La difficulté vient alors du fait que l'exécution des instructions du code adjoint doit correspondre à un

déroulement en sens inverse des instructions exécutées dans la phase progressive, et cela quelle que soit la valeur donnée aux variables indépendantes.

On pourra se convaincre que la manière de procéder suivante conviendra à diverses situations. Au cours de la phase progressive, on mémorise dans une *pile*, à chaque `goto` (y compris les `goto` implicites correspondant à l'accès à une instruction étiquetée), une étiquette repérant l'instruction exécutée avant ce `goto`. Cela pourra se faire au moyen d'une instruction `push(label)`. Au cours de l'exécution du code adjoint, il suffira alors de récupérer sur la pile, l'adresse d'où l'on est parti pour arriver à une étiquette donnée. Cela pourra se faire au moyen d'une instruction `pop(label)`. Comme exemple, considérons le code original suivant:

```
      instructions A
      goto 10
      instructions B
      goto 10
      instructions C
10    continue
      instructions D.
```

La phase progressive pourra s'exécuter comme suit:

```
      instructions A
      push (101)
      goto 10
      instructions B
      push (102)
      goto 10
      instructions C
      push (103)
10    continue
      instructions D.
```

et la phase régressive s'exécutera de la manière suivante:

```
      code adjoint de ( instructions D )
      pop (label)
      goto label
103   continue
      code adjoint de ( instructions C )
102   continue
      code adjoint de ( instructions B )
101   continue
      code adjoint de ( instructions A ).
```

Les **sous-routines** sont des structures particulières qui peuvent être traitées de deux manières différentes. Désignons par

$$f_s : I\!R^{n_s} \rightarrow I\!R^{m_s} : x_{P_s} \rightarrow x_{Q_s} = f_s(x_{P_s}),$$

la fonction réalisée dans une sous-routine donnée. La première possibilité est de calculer la jacobienne ∇f_s de cette fonction, ce qui peut se faire en utilisant le mode direct (par exemple

si $1 \simeq n_s \ll m_s$) ou inverse (par exemple si $n_s \gg m_s \simeq 1$). Cela permet de disposer de la valeur des dérivées partielles $\partial f_s / \partial x_i \in \mathbb{R}^{m_s}$, $i \in P_s$, dont la connaissance suffit pour dualiser l'instruction *vectorielle* $x_{Q_s} := f_s(x_{P_s})$ réalisée par la sous-routine. Cela se fera au moyen de formules analogues à celles que nous avons introduites en section 2.3. L'autre possibilité (c'est en fait la seule méthode mise en oeuvre dans les codes existant actuellement) est de dualiser toutes les instructions de la sous-routine comme si elles étaient insérées dans le code de calcul de f à l'endroit où elle est appelée.

Le choix entre ces deux possibilités dépendra de chaque cas. Par exemple, le premier choix s'imposera si $n_s = 1$ (mode direct) ou $m_s = 1$ (mode inverse) et que le calcul de f_s utilise beaucoup de variables locales intervenant de façon non linéaire. En effet, dans ce cas toutes les informations sur f_s nécessaires au calcul du gradient de f sont rassemblées dans ∇f_s et la valeur des variables locales peut être oubliée dès que le calcul de f_s et ∇f_s est terminé. On peut ainsi gagner de la place en mémoire. Par contre, on choisira la seconde possibilité si n_s et m_s sont tous deux $\gg 1$ et que la sous-routine utilise peu de variables locales intervenant de façon non linéaire dans le calcul de f_s. En effet, dans ce cas le calcul de la jacobienne ∇f_s peut être coûteux sans que l'on ne gagne en place mémoire. Il semble raisonnable de pouvoir détecter ces deux cas extrêmes de façon automatique. En dehors de ceux-ci, l'assistance du programmeur sera utile.

5 Quelques réalisations

Cette section contient une description succincte des différentiateurs automatiques que nous avons testés: JAKEF, GRESS, PADRE2 et ADOL-C. Des informations plus complètes pourront être obtenues par la lecture des références mentionnées dans le texte.

5.1 JAKEF

JAKEF est un précompilateur FORTRAN dans le sens où il accepte en entrée une *sous-routine* écrite en FORTRAN et fournit en sortie une sous-routine écrite également en FORTRAN. La sous-routine d'entrée est supposée calculer une fonction $f : \mathbb{R}^n \to \mathbb{R}^m$ et la sous-routine de sortie est à même de calculer à la fois la fonction f et sa jacobienne ∇f (ou son gradient si $m = 1$).

JAKEF a été développé à l'Argonne National Laboratory par K.E. Hillstrom (1985) à partir du précompilateur JAKE écrit par Speelpenning (1980). JAKE a été écrit en C pour des raisons de commodité de ce langage pour l'écriture de compilateurs. Par contre, JAKEF est écrit en FORTRAN. Le choix de ce langage est principalement motivé par le besoin d'avoir un module de différentiation automatique dans la bibliothèque MINPACK (routines d'optimisation en FORTRAN) d'Argonne.

JAKEF est essentiellement conçu pour calculer des dérivées premières. On pourrait penser que des dérivées d'ordre supérieur pourraient être obtenues par application répétitive du précompilateur. Il n'en est rien. En effet, une limitation importante de ce code est de ne pas pouvoir accepter en entrée des sous-routines appelant elles-mêmes d'autres sous-routines. Or le code fourni en sortie du précompilateur contient des appels à des sous-routines de bibliothèque et ne peut donc plus être offert en entrée au précompilateur.

JAKEF a été écrit pour tester la réalisabilité d'un précompilateur pour la différentiation automatique en mode inverse et ne fonctionne que sur ce mode. S'il s'agit de calculer la

jacobienne d'une fonction vectorielle $f : I\!R^n \rightarrow I\!R^m$, il le fait ligne par ligne, en calculant le gradient de chacune des m composantes de la fonction. Il n'est donc pas conseillé de l'utiliser lorsque $m \gg n$. En effet, nous avons vu que dans ce cas, il était préférable d'utiliser le mode direct de différentiation.

JAKEF fonctionne pour des fonctions obéissant au modèle étendu (voir section 2), c'est-à-dire qu'il n'introduit pas de variables intermédiaires pour se ramener au modèle simple. Cependant, toutes les fonctions d'affectation intervenant dans le calcul des variables dépendantes sont décomposées en fonctions élémentaires n'ayant que 0, 1 ou 2 opérandes. Cela nécessite l'introduction d'un certain nombre de variables intermédiaires. L'implémentation du mode inverse se fait au moyen d'une base de données qui est l'image du graphe de calcul de la fonction à différentier.

On pourra trouver des informations complémentaires sur JAKEF dans la thèse de Speelpenning (1980) et dans le manuel d'utilisation de Hillstrom (1985). Notons que la maintenance de JAKEF n'est plus assurée. La raison vient de ce qu'un autre logiciel, ADOL-C, a été développé à Argonne. Nous y reviendrons en section 5.4

5.2 GRESS

GRESS (Gradient Enhanced Software System) est aussi un précompilateur transformant un programme FORTRAN en un autre programme FORTRAN. Il a été développé à l'Oak Ridge National Laboratory, autour de J.E. Horwedell.

Ce précompilateur est opérationnel sur Machines VAX/VMS uniquement. La particularité de GRESS est d'utiliser un "pseudo-code" conjointement avec le programme précompilé au moment de l'exécution. Dans le programme de sortie, chaque opération est stockée dans une table et référencée par un pointeur. La présence de ce pseudo-code rend l'outil dépendant des machines VAX; de plus la génération du précompilateur fait appel à des programmes écrits en C et en FORTRAN.

GRESS connait les modes direct et inverse de la différentiation automatique. Il permet d'analyser des sous-programmes faisant appel à d'autres sous-programmes ou des boucles emboîtées. Enfin, GRESS permet d'analyser aussi bien des programmes principaux que des sous-programmes.

On peut faire remarquer que l'espace mémoire requis est très important, même pour des programmes originaux de petite taille (par exemple, le traitement du cas-test MMJOSR, 6 fonctions et 6 variables, a nécessité 8 Mo de mémoire vive !). Ceci rend l'usage de GRESS problématique dans les cas, fréquents, de taille plus importante. L'utilisation de GRESS en mode direct reste relativement simple mais en mode inverse, la suite des opérations et le relevé de résultats intermédiaires, à la main, devant servir à des calculs ultérieurs, constituent une tâche assez fastidieuse.

Pour conclure, l'utilisation de GRESS s'est en général avérée nettement moins simple que celle de JAKEF ou PADRE2.

5.3 PADRE2

Comme JAKEF, PADRE2 ([31, 32]) est un précompilateur FORTRAN dans le sens où il accepte en entrée une fonction écrite en FORTRAN et fournit en sortie une fonction écrite également en FORTRAN. Le programme d'entrée est supposé calculer une fonction $f : I\!R^n \rightarrow I\!R^m$ et le

programme de sortie est à même de calculer à la fois la fonction f et sa jacobienne ∇f (ou son gradient si $m = 1$).

PADRE2 a été développé à l'université de Tokyo par M. Iri et K. Kubota et constitue une extension de PADRE initialement développé par Iwata ([34]). PADRE2 a été écrit en C au moyen de l'analyseur syntaxique YACC. Il est disponible sur machine UNIX et sur PC sous MS-DOS, auprès des auteurs. Il permet le calcul des dérivées premières et secondes d'une fonction scalaire ou vectorielle et fournit deux types possibles d'estimation de l'erreur d'arrondi commise lors du calcul de la fonction. Il s'agit donc là d'une caractéristique intéressante de ce précompilateur. PADRE2 peut fonctionner en mode direct ou en mode inverse, seul ce dernier mode fournissant naturellement une estimation de l'erreur d'arrondi. PADRE2 possède certaines limitations par rapport au jeu d'instructions du FORTRAN standard, décrites dans le manuel de référence ([32]); signalons que ce dernier est suffisamment clair pour permettre une utilisation rapide de PADRE2. Enfin, PADRE2 ne peut traiter des sous-programmes imbriqués (appel à des sous-programmes), tout comme JAKEF; néanmoins des développements en ce sens sont annoncés dans le manuel de référence.

5.4 ADOL-C

ADOL-C a été développé par A. Griewank et ses collaborateurs [26]. C'est un différentiateur automatique très complet, capable, notamment, de calculer des dérivées d'ordre quelconque.

ADOL-C est basé sur le principe de la surcharge des opérateurs (section 4.2) et est écrit en C++. L'ensemble du package est constitué de fichiers d'entête à inclure dans les sous-routines faisant appel aux structures définies par ADOL-C, ainsi que d'une bibliothèque définissant les opérateurs et routines surchargés. Partant, comme ce fut notre cas, d'un code écrit en FORTRAN, la première chose à faire est de le traduire en C++. Cette tâche peut être réalisée automatiquement. Pour notre part, nous avons fait cette traduction au moyen du convertisseur `f2c` de AT&T [13], qui suppose que le code est écrit en FORTRAN 77. Notons qu'il est aussi possible de ne traduire en C++, que la partie du code calculant la fonction et de faire les appels aux sous-routines d'accompagnement à partir du code FORTRAN. Cette opération est délicate étant donné la différence de convention entre compilateurs.

Ses caractéristiques principales sont reprises dans la table 1.

5.5 Comparaison des fonctionnalités

Nous résumons sous forme de tableau (Table 1) les caractéristiques et les fonctionnalités implémentées dans les trois différentiateurs testés: JAKEF, GRESS, PADRE2 et ADOL-C. Dans ce tableau, le signe '+' signifie que la fonctionnalité est présente, le signe '−' signifie qu'elle est absente. Voici quelques indications sur les différentes entrées du tableau.

Langage du code: Langage dans lequel le code est écrit.

Langage traité: Langage dans lequel doit être écrit le programme calculant la fonction. Rappelons que, pour l'utilisation d'ADOL-C, l'utilitaire `f2c` permet de convertir un code écrit en FORTRAN en un code C++.

Modèle de programmes: Spécifie si le différentiateur se ramène au modèle simple de programmes (voir Section 2.1) en introduisant des variables auxiliaires. Ce cas est étiqueté par le mot 'simple', sinon le mot 'étendu' est utilisé.

Sélection d'instructions: Par cette fonctionnalité, nous entendons la capacité de repérer les instructions n'intervenant pas dans le calcul de la fonction et donc de ne pas générer de code pour ces instructions. Pour GRESS ce repérage doit être fait de façon manuelle, en introduisant des fanions (`*EXAP ON/OFF`) aux endroits appropriés, avant l'étape de précompilation.

Ordre de dérivation: Un ordre t signifie que le précompilateur peut générer le code nécessaire au calcul des dérivées t-ième. Un ordre ∞ signifie que t peut être pris aussi grand que la mémoire le permet. Dans le cas d'une fonction scalaire f, PADRE2 peut calculer le produit $\nabla^2 f(x)h$ pour $h \in I\!R^n$. En faisant varier h, on peut alors obtenir la hessienne de f.

Groupement de variables: Si cette fonctionnalité est présente, cela signifie qu'il est possible de placer les variables indépendantes ou dépendantes dans plusieurs variables ou tableaux.

Instructions conditionnelle, répétitive: Les instructions conditionnelles sont les instructions du type `if-then-else`. Les instructions répétitives sont les instructions `do` en FORTRAN et les instructions `for`, `while`, `do-while` en C.

Appel de sous-routines: Avec cette fonctionnalité, le calcul des fonctions pourra se faire en utilisant des appels en cascade à des sous-routines écrites par l'utilisateur.

Caractéristiques	JAKEF	GRESS	PADRE2	ADOL-C
langage du code	FORTRAN	C, FORTRAN	C	C++
langage traité	FORTRAN	FORTRAN	FORTRAN	C++
mode direct	−	+	+	+
mode inverse	+	+	+	+
modèle de programmes	étendu	étendu	simple	simple
sélection d'instructions	+	−	?	+
ordre de dérivation	1	1	2	∞
estimation d'erreur	−	−	+	−
groupement de variables	+	+	−	+
variables en simple precision	+	+	+	−
variables en double precision	+	+	−	+
instruction conditionnelle	+	+	+	+
boucle répétitive	+	+	+	+
appel de sous-routines	−	+	−	+
appel récursif de sous-routines	−	−	−	+

Table 1: Comparaison des caractéristiques de JAKEF, GRESS, PADRE2 et ADOL-C.

6 Tests comparatifs

Les tests que nous présentons ci-après nous ont également servi à valider les différentiateurs quant au calcul correct du gradient dans des situations variées. Nous avons en effet vérifié dans chaque cas que les réponses fournies par les codes étaient identiques. De plus, pour tous

les problèmes-test nous disposions d'un programme de calcul du gradient, ce qui a permis des vérifications ponctuelles supplémentaires.

Bien évidemment, les mots-clés JAKEF, GRESS, PADRE2 et ADOL-C désignent les codes du même nom, dont la description est donnée aux sections 5.1, 5.2, 5.3 et 5.4. Le mot-clé DD fait référence à des dérivées calculées par différences divisées. Le mot-clé MANUEL désignera un code de calcul de dérivées écrit manuellement, par un numéricien, suivant une méthode qu'il n'a pas toujours été possible d'identifier. Enfin, METEO caractérisera des résultats obtenus par la méthode du code adjoint (Section 4.1.2, [67] et [10]).

Les tests avec JAKEF, PADRE2 et ADOL-C ont été réalisés sur une SPARCstation 1 avec 8 Méga-octets de mémoire centrale (un nombre `real` ou `integer` tient sur 32 bits) et ceux avec GRESS sur un VAX 8250 avec 32 Mo de mémoire, un réel étant codé sur 32 bits.

En ce qui concerne la place mémoire, il s'agira toujours de la place à réserver *en plus* de celle déjà occupée par le reste du code, pour pouvoir calculer le gradient par la méthode en question. Nous avons toujours compté la place occupée par le vecteur devant recevoir le gradient. En plus de ce vecteur, nous n'avons compté pour JAKEF, que l'espace occupé par les variables `ygrad`, `ifs` et `rfs` et pour PADRE2, que celui occupé par les variables `node`, `value` et `namtbl`. Pour ADOL-C, nous avons compté la place occupée par l'arbre de calcul (qui en général est sauvegardé sur disque, `bufsize` = 1024) et par ce qui semble être les variables duales *en simple précision* (`revreal` = `float` dans le fichier d'entête `usrparms.h`). Ces deux valeurs s'obtiennent en octet par les arguments `length` et `deaths` de la sous-routine `tapestats` et valent (`length`) et (4 `deaths`), respectivement.

Certains résultats de cette section ont été présentés dans [17].

6.1 Traitement d'un processus itératif: MMJOSR

Le sous-programme MMJOSR permet de passer d'une représentation képlérienne d'une orbite à une représentation cartésienne et calcule la jacobienne de la transformation. Pour cela, il est nécessaire de résoudre l'équation de Képler pour le calcul de l'anomalie excentrique (équation implicite): ceci est fait par une méthode de Newton. Ce cas présente donc deux caractéristiques intéressantes pour l'analyse des précompilateurs: d'une part, on est confronté au problème d'appel à des fonctions externes: comme nous l'avons mentionné au paragraphe 5.2, GRESS possède les fonctionnalités pour traiter le cas d'appels de fonctions externes (ADOL-C les a également). D'autre part, la méthode de Newton utilisée pour résoudre l'équation de Képler donne un exemple de processus itératif. Le traitement complet de ce cas-test n'a été mené qu'au moyen de GRESS. Les résultats obtenus apparaissent dans les tableaux 2 et 3.

.120088D+01	-.161028D+07	-.131209D+07	-.220804D+08	-.223492D+08	0.757021D+05
.915949D+00	0.404274D+08	0.837490D+06	0.292430D+08	0.293016D+08	0.138382D+08
.637946D-01	0.283799D+07	-.126745D+08	0.421807D+07	0.000000D+00	0.906544D+06
.256962D-06	0.639218D+03	0.126578D+03	-.228501D+04	-.229226D+04	-.140508D+04
.469724D-04	-.293851D+04	-.807936D+02	-.298339D+01	0.125397D+02	-.107169D+04
.307716D-05	-.260339D+03	0.122273D+04	0.236351D+03	0.000000D+00	0.746422D+02

Table 2: MMJOSR: résultats explicites

.12008D+01	-.161028D+07	-.131209D+07	-.220804D+08	-.223492D+08	0.757020D+05
.915949D+00	0.404274D+08	0.837490D+06	0.292430D+08	0.293016D+08	0.138382D+08
.637946D-01	0.283799D+07	-.126745D+08	0.421807D+07	0.000000D+00	0.906544D+06
.256963D-06	0.639218D+03	0.126578D+03	-.228501D+04	-.229226D+04	-.140508D+04
.469724D-04	-.293851D+04	-.807936D+02	-.298339D+01	0.125397D+02	-.107169D+04
.307716D-05	-.260339D+03	0.122273D+04	0.236351D+03	0.000000D+00	0.746422D+02

Table 3: MMJOSR: résultats par GRESS

On peut constater, au vu des résultats de ces deux tableaux, que les calculs effectués par GRESS sont corrects; mentionnons pour plus de précision que l'erreur relative maximale commise dans le calcul de la jacobienne est de 10^{-6}.

Pour l'évaluation de PADRE2 sur le cas d'un processus itératif, nous avons utilisé la fonction qui résout par une méthode de Newton l'équation de Képler, en l'anomalie excentrique. Cette équation s'écrit: $M = E - e \sin E$, où E est l'anomalie excentrique, e l'excentricité et M l'anomalie moyenne. On obtient donc E en sortie du programme ainsi que $\partial E/\partial e$ et $\partial E/\partial M$. Ces deux dernières dérivées s'obtiennent simplement par:

$$\partial E/\partial e \quad = \quad \frac{\sin E}{1 - e \cos E}$$

et

$$\partial E/\partial M \quad = \quad \frac{1}{1 - e \cos E}$$

et valent respectivement, compte tenu des données du problème, 0.4719 et 0.6615. Les résultats obtenus au moyen de PADRE2 sont les suivants:

$$\begin{aligned} \text{Anomalie excentrique} &= 2.34733 \\ \text{Erreur d'arrondi (anomalie)} &= 2.73538\,10^{-7} \\ \text{Gradient} &= (0.471894,\ \ 0.661519). \end{aligned}$$

On constate que les résultats sont conformes; nous avons en outre illustré la possibilité d'obtenir une estimation de l'erreur commise lors du calcul de E (il s'agit ici d'une borne absolue sur l'erreur).

6.2 Un problème en météorologie: U1MT1

Le cas-test U1MT1 provient de la bibliothèque MODULOPT de l'INRIA et fait partie d'une batterie de problèmes-test en optimisation. Ce programme détermine les vitesse et pression atmosphériques à partir d'un modèle météorologique bidimensionnel statique et de mesures des vitesse et pression atmosphériques en divers points. Une méthode de moindres carrés et de lagrangien augmenté est utilisée et donc la fonction calculée est à valeurs dans $I\!R$.

Ce problème a été retenu pour les raisons suivantes.

– C'est un problème de grande taille (1875 variables indépendantes), provenant d'un modèle physique réaliste.

- Le calcul de la fonction se fait sans appel de sous-routine, ce qui permet un traitement par JAKEF et PADRE2.
- On dispose d'un programme de calcul du gradient, ce qui permettra de vérifier les résultats produits par différentiation automatique.

Il nous a paru intéressant de faire également, pour ce cas-test, une comparaison avec le calcul du gradient par différences divisées (DD). L'utilisation de GRESS pour cette fonction n'a pas été possible étant donné que U1MT1 a plus des 200 variables qu'il autorise.

Le tableau de la table 4 résume les résultats obtenus sur une SPARCstation 1. La *mémoire supplémentaire requise* donne le nombre d'entiers (`integer`), de réels (`real`) et le total de ces deux nombres convertis en kilo-octet, qu'il faut réserver en plus de l'espace déjà utilisé pour le calcul de la fonction seule. Celui-ci se fait en utilisant 34.3 Ko de données. Les *écarts relatifs* sont définis par:

$$\textit{écart relatif maximal} := \max_{1 \le i \le n} \frac{|\tilde{g}_i - g_i|}{|g_i|},$$

$$\textit{écart relatif moyen} := \frac{1}{n} \sum_{i=1}^{n} \frac{|\tilde{g}_i - g_i|}{|g_i|},$$

lorsque g_i est non nul pour tout i. Ici, g représente le gradient calculé par la méthode manuelle et $\tilde{g}$ est le gradient calculé par DD, JAKEF, PADRE2, ADOL-C ou METEO.

Type de calcul	mémoire supplémentaire requise			temps	écart relatif	
	`integer`	`real`	total (Ko)	CPU (sec)	maximal	moyen
f seul	—	—	—	0.04	—	—
f et g manuel	5	1923	7.5	0.23	—	—
f et g par DD	—	—	—	83.86	$1.\ 10^{+3}$	$3.\ 10^{-1}$
f et g par JAKEF	144073	148398	1142.4	1.19	$9.\ 10^{-5}$	$6.\ 10^{-7}$
f et g par PADRE2	211792	133393	1348.3	1.14	$9.\ 10^{-5}$	$6.\ 10^{-7}$
par ADOL-C:	—	—	1362.4	—	—	—
f avec graphe	—	—	—	4.75	—	—
f sans graphe	—	—	—	2.24	—	—
g avec graphe	—	—	—	5.50	$3.\ 10^{-2}$	$3.\ 10^{-5}$
g sans graphe	—	—	—	3.14	$3.\ 10^{-2}$	$3.\ 10^{-5}$
g par METEO	10	3796	14.8	0.09	$9.\ 10^{-5}$	$7.\ 10^{-7}$

Table 4: Comparaison de performances sur le cas-test U1MT1

En ce qui concerne la place mémoire, on voit que celles utilisées par JAKEF, PADRE2 et ADOL-C s'équivalent, mais sont beaucoup plus importantes que celle utilisée par la méthode manuelle. Quant à la méthode METEO, elle est particulièrement sobre: en gros, elle ne demande que 2 vecteurs supplémentaires, dont l'un contient le gradient. On sait (section 4.1.2) que cet espace mémoire dépend du choix de mémoriser ou de recalculer certaines variables intermédiaires intervenant de façon non-linéaire dans le code. Nous avons choisit de ne mémoriser qu'un seul vecteur de dimension $n = 1875$ et de recalculer toutes les autres variables.

Les temps de calcul ont été évalués en faisant une moyenne sur 10 essais (sauf pour DD). Le temps de calcul par différences divisées est considérable. Il dépasse n fois le temps de calcul de f seule, parce que, pour certaines composantes, l'accroissement dx_i est réadapté et la différence recalculée. Les temps réalisés par JAKEF et PADRE2 sont comparables et inférieurs aux temps d'ADOL-C. Pour ce dernier, le temps correspondant à "f avec graphe" est essentiellement le temps nécessaire à construire une représentation du graphe de calcul de la fonction, ceux de "f sans graphe" (sous-routine `forward`) et de "g sans graphe" (sous-routine `reverse`) correspondent au calcul de la fonction et de son gradient g (en mode inverse) en utilisant le graphe construit précédemment. Enfin "g avec graphe" est le temps de la sous-routine `grad_eval`, construisant le graphe et évaluant le gradient. Le temps de calcul utilisé par ADOL-C peut en partie s'expliquer par le fait que certains calculs sont faits en double précision, alors que tout est fait en simple précision pour les autres codes.

Sur SUN 3/60, les temps sont plus importants, ce qui permet de les comparer raisonnablement entre eux. On observe que le temps de calcul de f et ∇f par JAKEF et PADRE2 (8.76 ...9.36 sec) est approximativement égal à 8 fois celui du calcul de f (1.11 sec). L'estimation $C'_{\mathcal{F}} \leq 5$ obtenue en (34) n'est donc pas satisfaite. Cela doit être attribué au coût de mise en oeuvre informatique du mode inverse dans ces codes, puisque pour la méthode du code adjoint (1.73 secondes pour le gradient seul), qui est également un mode inverse, le quotient de (33) vaut 2.6. Le temps mis par cette dernière méthode est, on le constate, tout à fait saisissant puisqu'il vaut la moitié de celui du code manuel et cela malgré notre choix de recalculer pratiquement toutes les variables intermédiaires !

La faible précision obtenue par différences divisées peut paraître surprenante. Elle nous avait d'abord fait penser à une erreur dans le code de calcul du gradient. C'est en fait le résultat obtenu avec les différentiateurs automatiques qui nous a fait revenir sur cette impression ! Dans cette méthode, l'accroissement dx_i de la coordonnées x_i est déterminé par la formule

$$dx_i = \begin{cases} 10^{-5} \max\left(|x_i|, \left|\dfrac{f}{g_i}\right|\right) & \text{si} \quad g_i \neq 0 \\ 10^{-5}|x_i| & \text{si} \quad g_i = 0. \end{cases}$$

Lorsque ce calcul donne $dx_i = 0$, on prend $dx_i = 10^{-5}$. Le choix de 10^{-4} au lieu de 10^{-5} dans la formule ci-dessus a pour résultat de diminuer l'écart relatif maximal (qui passe à $2.\ 10^{+2}$) mais d'augmenter l'écart relatif moyen (qui passe à $5.\ 10^{-1}$).

Nous avons vérifié que les écarts relatifs entre la méthode manuelle et JAKEF diminuaient lorsque l'on augmentait la précision des calculs: le passage à la double précision a été simulé en faisant tourner les codes sur un CRAY 2 (un réel est représenté par 32 bits sur une SPARCstation 1 et par 64 bits sur le CRAY 2). Les écarts relatifs maximal et moyen deviennent respectivement $6.\ 10^{-12}$ et $4.\ 10^{-14}$. De ce résultat, on peut conclure que l'écart entre les gradients calculés manuellement et JAKEF sont dus aux erreurs d'arrondi et non à une erreur dans l'une ou l'autre méthode. D'autre part, JAKEF et PADRE2 donnent exactement les mêmes valeurs pour le gradient: les écarts relatifs sont nuls ! En fait, à part une question de représentation, les méthodes sont identiques.

6.3 Un cas d'école: boucles emboîtées

Ce cas-test sert essentiellement à mettre en évidence le problème de place mémoire que pose certains différentiateurs automatiques fonctionnant sur le mode inverse. Ce problème est

actuellement une des difficultés majeures freinant l'utilisation de ces différentiateurs pour les grands problèmes.

Il s'agit d'une sous-routine calculant $f(x) = (\texttt{nfois})^3\|x\|^2$ d'une manière un peu laborieuse: on utilise trois boucles emboîtées dont les indices de boucles varient de `1` à `nfois` (paramètre ajustable). La norme de x au carré est calculée dans la boucle la plus interne.

```
      subroutine boucle (n,nfois,x,f)
      integer n,nfois
      real x(n),f
      integer i,j,k,l
      real r
      f=0.
      do 40 i=1,nfois
         do 30 j=1,nfois
            do 20 k=1,nfois
               do 10 l=1,n
                  r=x(l)
                  f=f+r*r
10             continue
20          continue
30       continue
40    continue
      return.
```

Pour ce cas d'école, pourtant simple, la place mémoire demandée par JAKEF et PADRE2 est proportionnelle à $(\texttt{nfois})^3$ et devient rapidement trop importante. Notons qu'on aurait observé le même problème avec ces différentiateurs pour une sous-routine plus simple encore que `boucle`: celle où on y remplace la boucle la plus interne, indicée par `l`, par l'instruction `f=0.` ! Si nous avons pris un exemple à peine plus complexe, c'est pour montrer que la méthode du code adjoint n'a, elle, aucune difficulté avec ce type de situation. Le code adjoint peut en effet s'écrire comme suit (le gradient est renvoyé dans `xd`):

```
      subroutine bouclm (n,nfois,x,xd)
      integer n,nfois
      real x(n),xd(n)
      integer i,j,k,l
      real r
c
c --- variables duales
c
      real rd,fd
c
c --- initialisation des variables duales
c
      fd=1.
      rd=0.
```

```
      do 1 i=1,n
         xd(i)=0.
   1  continue

c
c --- c'est parti
c
      do 40 i=nfois,1,-1
         do 30 j=nfois,1,-1
            do 20 k=nfois,1,-1
               do 10 l=n,1,-1
                  r=x(l)
                  rd=rd+2.*r*fd
                  xd(l)=xd(l)+rd
                  rd=0.
   10          continue
   20       continue
   30    continue
   40 continue
      return.
```

Le tableau de la table 5 donne les résultats de nos essais pour $n = 10$ et `nfois` variant de 5 à 25. On y a noté "Ko" pour la place mémoire en kilo-octet, "sec" pour le temps de calcul CPU en seconde et "$C_{\mathcal{F}}$" pour le rapport $T(f, \nabla f)/T(f)$. On observe que pour

	f	JAKEF			PADRE2			METEO	
`nfois`	sec	Ko	sec	$C_{\mathcal{F}}$	Ko	sec	$C_{\mathcal{F}}$	sec	$C_{\mathcal{F}}$
5	0.01	78.	0.08	8.0	49.	0.04	4.0	0.01	2.0
10	0.03	625.	0.64	21.3	390.	0.38	12.7	0.04	2.3
15	0.10	2109.	2.19	21.9	1318.	1.25	12.5	0.13	2.3
20	0.21	5000.	5.24	25.0	3125.	3.20	15.2	0.32	2.5
25	0.41	9765.	10.41	25.4	6103.	7.21	17.6	0.62	2.5

Table 5: Comparaison de performances sur les "boucles emboîtées"

JAKEF et PADRE2, la mémoire nécessaire s'acroît comme $(\texttt{nfois})^3$. On voit aussi que, pour ces mêmes codes, le quotient $C_{\mathcal{F}}$ se détériore lorsque `nfois` augmente. Ceci ne peut être dû qu'au surcoût occasionné par la gestion du graphe de calcul et non pas au mode inverse. On voit en effet que, pour la méthode METEO (tout au moins pour notre implementation du code adjoint), $C_{\mathcal{F}}$ reste inférieur à la limite théorique de 5 (voir Section 2.3.9). D'ailleurs, si on ne compte que les opérations effectuées dans la boucle la plus interne de `bouclm`, on a pour la méthode METEO:

$$C_{\mathcal{F}} \simeq \frac{7T(\boxplus\!\!\rightarrow) + 6T(\boxplus\!\!-) + 3T(+) + 3T(*)}{3T(\boxplus\!\!\rightarrow) + 2T(\boxplus\!\!-) + T(+) + T(*)} \leq 3,$$

où $T(\boxminus\!\!\rightarrow)$ et $T(\boxminus\!\!-)$ représentent respectivement les temps qu'il faut pour aller chercher et pour sauvegarder un nombre flottant en mémoire. Cette borne sur $C_{\mathcal{F}}$ est donc valable pour n et `nfois` grands. D'après le tableau 5, elle est aussi satisfaite pour n et `nfois` petits.

Pour des essais de différentiateurs automatiques sur un grand problème réel ayant des boucles emboîtées, problème dont nous nous sommes inspirés pour introduire le cas d'école ci-dessus, on lira avec intérêt les expériences de E. Soulié (1991).

7 Conclusion

Nous avons dans cette étude parcouru plusieurs aspects de la différentiation automatique de fonctions représentées par des programmes, allant de ses fondements théoriques jusqu'à l'étude et la mise à l'épreuve de quelques réalisations informatiques à présent disponibles. Ceci nous a conduit au recensement des techniques d'implémentation. Nous avons également regardé comment ces méthodes pouvaient être utilisées dans le domaine particulier de l'analyse des erreurs d'arrondi dans les programmes de calcul numérique.

Il ressort de ce travail que la différentiation automatique est un outil précieux, dont l'utilité et le champ d'application iront sans aucun doute en s'accroissant. Les possibilités sont en effet diverses et souvent absentes des autres approches de la différentiation assistée. Cette diversité et le fait que la différentiation automatique n'en est qu'à ses débuts sont des gages d'une activité prometteuse. Ceci veut aussi dire qu'il paraît peu probable de voir dans l'immédiat un différentiateur automatique universel, convenant à toutes les situations nécessitant le calcul de dérivées. En effet, à des besoins différents (calcul de gradient ∇f, de jacobienne ∇F, de dérivées d'ordre supérieur, gestion serrée de la place mémoire, analyse des imprécisions numériques, ...) correspondent des modes de différentiation et des manières de générer un code adjoint différents. Ce n'est sans doute qu'au cours des années et après une réflexion approfondie sur la structure des besoins, que les codes actuels pourront évoluer vers des solutions simples et souples.

Les tests numériques que nous avons faits ont également clairement établi les atouts de la différentiation automatique par écriture d'un code adjoint (Section 4.1.2). Comme cette technique n'est pas actuellement automatisée, nous espérons que ce rapport servira à stimuler les initiatives dans ce sens. Cela nécessitera une collaboration étroite entre informaticiens et numériciens.

Notons pour terminer que les actes du colloque "1991 SIAM Worshop on Automatic Differentiation of Algorithms: Theory, Implementation and Application", s'étant tenu entre les 3 et 8 janvier 1991 à Breckenridge (Colorado), pourront servir de complément à cette étude.

A Transformation des termes d'une série de Taylor

Soit ξ une fonction réelle définie et de classe C^t dans le voisinage d'un point $u \in I\!R^r$. On note sa dérivée d'ordre t ($t \geq 0$) en $u \in I\!R^r$ dans une direction $\bar{u} \in I\!R^r$ par:

$$\xi^{(t)} \equiv \xi^{(t)}(u) \cdot (\bar{u})^t,$$

tandis que le $(t+1)$-ième terme du développement de Taylor de $\xi(u+\bar{u})$ autour de u se note:

$$\xi^{[t]} \equiv \frac{1}{t!}\, \xi^{(t)}(u) \cdot (\bar{u})^t.$$

La table suivante (*cf.* [40, Section 4.5], [51, Section 3.4] et [57]) donne les formules de transformation des termes du développement en série de Taylor, lors de quelques opérations élémentaires. Dans cette table, x_i, x_j, x_k et x_l sont des fonctions réelles définies et régulières dans un voisinage de u.

Opération	Tranformation des termes de Taylor
$x_k = x_i \pm x_j$	$x_k^{[t]} = x_i^{[t]} \pm x_j^{[t]}, \quad t \geq 0$
$x_k = x_i\, x_j$	$x_k^{[t]} = \sum_{s=0}^{t} x_i^{[s]}\, x_j^{[t-s]}, \quad t \geq 0$
$x_k = \dfrac{x_i}{x_j}$	$x_k^{[t]} = \dfrac{1}{x_j^{[0]}} \left[x_i^{[t]} - \sum_{s=0}^{t-1} x_k^{[s]}\, x_j^{[t-s]} \right], \quad t \geq 1$
$x_k = \exp(x_i)$	$x_k^{[t]} = \dfrac{1}{t} \sum_{s=1}^{t} s\, x_k^{[t-s]}\, x_i^{[s]}, \quad t \geq 1$
$x_k = \log(x_i)$	$x_k^{[1]} = \dfrac{x_i^{[1]}}{x_i^{[0]}}$ $x_k^{[t]} = \dfrac{1}{x_i^{[0]}} \left[x_i^{[t]} - \dfrac{1}{t} \sum_{s=1}^{t-1} s\, x_k^{[s]}\, x_i^{[t-s]} \right], \quad t \geq 2$
$x_k = \sin x_i$ $x_l = \cos x_i$	$x_k^{[t]} = \dfrac{1}{t} \sum_{s=1}^{t} s\, x_l^{[t-s]}\, x_i^{[s]}$ $x_l^{[t]} = -\dfrac{1}{t} \sum_{s=1}^{t} s\, x_k^{[t-s]}\, x_i^{[s]}$ $t \geq 1$
$x_k = \arctan x_i$ $x_l = \dfrac{1}{1+x_i^2}$	$x_k^{[t]} = \dfrac{1}{t} \sum_{s=1}^{t} s\, x_l^{[t-s]}\, x_i^{[s]}, \quad t \geq 1$

Table 6: Transformation des termes de la série de Taylor

A titre d'illustration, montrons comment s'obtiennent les formules pour le produit $x_k = x_i\, x_j$:

$$x_k^{[t]} = \sum_{s=0}^{t} x_i^{[s]}\, x_j^{[t-s]}.$$

Elles se vérifient par récurrence sur t. La formule pour $t = 0$ est vérifiée puisqu'elle s'écrit $x_k^{[0]} = x_i^{[0]}\, x_j^{[0]}$. Supposons ensuite que la formule soit vraie pour t, c'est-à-dire que l'on ait

$$x_k^{(t)} = \sum_{s=0}^{t} \binom{t}{s} x_i^{(s)}\, x_j^{(t-s)}.$$

En dérivant cette formule, on obtient

$$\begin{aligned} x_k^{(t+1)} &= \sum_{s=0}^{t} \binom{t}{s} x_i^{(s)}\, x_j^{(t+1-s)} + \sum_{s=0}^{t} \binom{t}{s} x_i^{(s+1)}\, x_j^{(t-s)} \\ &= x_i^{(0)}\, x_j^{(t+1)} + \sum_{s=1}^{t} \left[\binom{t}{s} + \binom{t}{s-1} \right] x_i^{(s)}\, x_j^{(t+1-s)} + x_i^{(t+1)}\, x_j^{(0)} \\ &= \sum_{s=0}^{t+1} \binom{t+1}{s} x_i^{(s)}\, x_j^{(t+1-s)}, \end{aligned}$$

où on a utilisé la relation du triangle de Pascal. On retrouve donc la formule précédente à l'ordre $t + 1$.

Remarquons que tous les seconds membres intervenant dans les formules de la table 6 sont (et doivent être) homogènes de degré t en $\bar{u}$. Cette propriété permet dans certain cas de retrouver aisément les formules. Par exemple pour la formule du produit, il suffit de multiplier les deux développement de x_i et x_j: le terme d'ordre t du développement du produit est alors obtenu en regroupant les termes homogènes de degré t en $\bar{u}$.

On trouvera dans le livre de Rall (1981) des remarques sur la simplification des formules binaires de la table 6 lorsque l'une des opérandes est constante.

En première analyse, l'utilisation des formules de la table 6 pour le calcul des t premiers termes de Taylor demande un nombre d'opérations de l'ordre de $\mathcal{O}(t^2)$. Pour de grandes valeurs de t, il est intéressant d'effectuer ces opérations par des méthodes plus sophistiquées. L'idée la plus simple est basée sur l'observation que les sommes apparaissant dans cette table proviennent de la convolution de vecteurs. Alors, si on transforme ces vecteurs par la transformation de Fourier (rapide), les convolutions deviennent des produits de vecteurs, faciles à effectuer (les produits se font entre éléments correspondants). Le coût total de ces opérations est de l'ordre de $\mathcal{O}(t \log t)$: voir Brent et Kung (1978) et Blahut (1985).

Notons que les formules de la table 6 ne sont valables que pour le calcul de dérivées unidirectionnelles. Si au contraire, on se donne deux directions $\bar{u}^1$ et $\bar{u}^2$ dans $\mathbb{R}^r$, les dérivées bidirectionnelles secondes de $x_k = x_i\, x_j$ s'écrivent

$$\begin{aligned} x_k''(u) \cdot (\bar{u}^1, \bar{u}^2) &= x_i''(u) \cdot (\bar{u}^1, \bar{u}^2)\, x_j(u) + x_i'(u) \cdot \bar{u}^1\, x_j'(u) \cdot \bar{u}^2 \\ &\quad + x_i'(u) \cdot \bar{u}^2\, x_j'(u) \cdot \bar{u}^1 + x_i(u)\, x_j''(u) \cdot (\bar{u}^1, \bar{u}^2). \end{aligned}$$

Plus généralement, si on note $\mathcal{P}_t$ l'ensemble des t-uples $\sigma = (\sigma_1, \ldots, \sigma_t)$ qui sont des permutations de $\{1, \ldots, t\}$ et

$$x_k^{(\sigma_1, \ldots, \sigma_s)} \equiv x_k^{(s)}(u) \cdot (\bar{u}^{\sigma_1}, \ldots, \bar{u}^{\sigma_s}),$$

et si on convient que $x_i^{(\ldots, \sigma_s, \ldots)} = x_i$ si $s \notin \{1, \ldots, t\}$, alors on a pour tout $t \geq 0$:

$$x_k^{(1, \ldots, t)} = \sum_{s=0}^{t} \sum_{\sigma \in \mathcal{P}_t} \frac{1}{s!} \frac{1}{(t-s)!}\, x_i^{(\sigma_1, \ldots, \sigma_s)}\, x_j^{(\sigma_{s+1}, \ldots, \sigma_t)}.$$

Cette formule montre que pour calculer, $x_k^{(1,\ldots,t)}$, il faut avoir évalué auparavant

$$x_i^{(\sigma_1,\ldots,\sigma_s)}, \quad 0 \leq s \leq t, \quad \sigma \in \mathcal{P}_t.$$

De même pour x_j. Or, compte tenu de la symétrie, il y a 2^t éléments de ce type et donc leur nombre croît exponentiellement avec t. Ceci montre que ce calcul est beaucoup plus coûteux en temps et en espace mémoire que celui des dérivées unidirectionnelles.

Terminons cette annexe en donnant des formules permettant de retrouver les dérivées multidirectionnelles à partir des dérivées unidirectionnelles. On a pour $t \geq 2$:

$$\xi^{(t)}(u) \cdot (\bar{u}^1, \ldots, \bar{u}^t) = \frac{1}{2^{t-1} t!} \sum_{\substack{\tau \in \{0,1\}^t \\ \tau_1 = 0}} (-1)^{(\sum_{s=1}^t \tau_s)} \, \xi^{(t)}(u) \cdot \left(\sum_{s=1}^{t} (-1)^{\tau_s} \bar{u}^s \right)^t .$$

La somme du second membre contient 2^{t-1} termes. Donc le calcul d'*une* dérivée multidirectionnelle au moyen de cette formule demandera l'évaluation de 2^{t-1} dérivées unidirectionnelles. Dans le cas où $t = 2$ et ξ ne dépend que d'une variable, on retrouve la formule bien connue:

$$\bar{u}^1 \bar{u}^2 = \frac{1}{4} \left[(\bar{u}^1 + \bar{u}^2)^2 - (\bar{u}^1 - \bar{u}^2)^2 \right].$$

B Transformation des variables duales des termes d'une série de Taylor

Comme dans l'annexe A, on note $x_i^{[s]}$, le $(s+1)$-ième terme du développement de Taylor de x_i (considéré comme fonction d'une variable $u \in I\!R^r$) pour un déplacement $\bar{u} \in I\!R^r$:

$$x_i^{[s]} \equiv \frac{1}{s!} x_i^{(s)}(u) \cdot (\bar{u})^s,$$

où $x_i^{(s)}$ est la dérivée d'ordre s de x_i.

Fixons $t \geq 0$ et supposons que les variables $x_i^{[s]}$ $(0 \leq s \leq t,\ 1 \leq i \leq N)$ soient calculées par un certain programme, par exemple (45) ou (46). On note p_i^s la variable duale de $x_i^{[s]}$. La valeur effective de p_i^s dépendra de la fonction scalaire des variables $x_j^{[r]}$ dont on veut le gradient, comme celle utilisée dans (46), mais la transformation des variables duales ne dépendra que de la manière dont les variables primales $x_j^{[s]}$ sont transformées – voir section 2.4.2. En supposant que les transformations primales sont celles de la table 6, la table 7 ci-dessous donne les formules de transformation correspondantes pour les variables duales.

Dans cette table, la notation "$s = t-1, \ldots, 0$" signifie que la formule à laquelle elle est attachée doit être exécutée *dans l'ordre*, d'abord pour $s = t-1$, puis $s = t-2$, ..., jusqu'à $s = 0$.

Opération	Tranformation des variables duales
$x_k = x_i \pm x_j$	$p_i^r := p_i^r + p_k^r, \quad 0 \le r \le t$ $p_j^r := p_j^r \pm p_k^r, \quad 0 \le r \le t$
$x_k = x_i\, x_j$	$p_i^r := p_i^r + \sum_{s=r}^{t} x_j^{[s-r]} p_k^s, \quad 0 \le r \le t$ $p_j^r := p_j^r + \sum_{s=r}^{t} x_i^{[s-r]} p_k^s, \quad 0 \le r \le t$
$x_k = \dfrac{x_i}{x_j}$	$p_k^s := p_k^s - \dfrac{1}{x_j} \sum_{r=s+1}^{t} x_j^{[r-s]} p_k^r, \quad s = t-1, \ldots, 0$ $p_i^r := p_i^r + \dfrac{p_k^r}{x_j}, \quad 0 \le r \le t$ $p_j^r := p_j^r - \dfrac{1}{x_j} \left[\sum_{s=r}^{t} x_k^{[s-r]} p_k^s \right], \quad 0 \le r \le t$
$x_k = \exp(x_i)$	$p_k^s := p_k^s + \sum_{r=s+1}^{t} \dfrac{r-s}{r}\, x_i^{[r-s]} p_k^r, \quad s = t-1, \ldots, 0$ $p_i^0 := p_i^0 + x_k^{[0]} p_k^0$ $p_i^r := p_i^r + r \sum_{s=r}^{t} \dfrac{1}{s}\, x_k^{[s-r]} p_k^s, \quad 1 \le r \le t$
$x_k = \log(x_i)$	$p_k^s := p_k^s - \dfrac{s}{x_i} \left[\sum_{r=s+1}^{t} \dfrac{1}{r}\, x_i^{[r-s]} p_k^r \right], \quad s = t-1, \ldots, 1$ $p_i^r := p_i^r + \dfrac{1}{x_i} \left[p_k^r - \sum_{s=r+1}^{t} \dfrac{s-r}{s}\, x_k^{[s-r]} p_k^s \right], \quad 0 \le r \le t$
$x_k = \sin x_i$ $x_l = \cos x_i$	$\left.\begin{array}{l} p_k^s := p_k^s - \sum_{r=s+1}^{t} \dfrac{r-s}{r}\, x_i^{[r-s]} p_l^r \\ p_l^s := p_l^s + \sum_{r=s+1}^{t} \dfrac{r-s}{r}\, x_i^{[r-s]} p_k^r \end{array}\right\} \quad s = t-1, \ldots, 0$ $p_i^r := p_i^r + r \sum_{s=r}^{t} \dfrac{1}{s} \left[x_l^{[s-r]} p_k^s - x_k^{[s-r]} p_l^s \right], \quad 1 \le r \le t$

Table 7: Transformation des variables duales des termes d'une série de Taylor

Références

[1] F.L. Bauer (1974). Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11, 87–96.

[2] R.E. Blahut (1985). *Fast Algorithms for Digital Signal Processing.* Addison-Wesley, Reading.

[3] B.E. Bliss (1989). *Instrumentation of Fortran Programs for Automatic Roundoff Error Analysis and Performance Evaluation.* Master's thesis, Center for Supercomputing Research and Develppment, University of Illinois at Urbana-Champaign, Urbana, IL 61801-2932.

[4] G. Bonnet (1989). Étude de la sensibilité aux erreurs numériques (méthode de Miller – étude théorique). Rapport de Stage ENSEEIHT, Rapport CNES TE/IS/MS/MN/371, Centre National d'Études Spatiales, Toulouse, France.

[5] G. Bonnet (1990). Étude de la sensibilité aux erreurs numériques (évaluation de la méthode de Miller). Rapport de Stage ENSEEIHT, Rapport CNES TE/IS/MS/MN/206, Centre National d'Études Spatiales, Toulouse, France.

[6] R.P. Brent, H.T. Kung (1978). Fast algorithms for manipulating formal power series. *Journal of the Association for Computing Machinery*, 25, 581–595.

[7] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, S.M. Watt (1988). Maple reference manuel. Symbolic Computation Group, Department of Computer Science, University of Waterloo, Ontario, Canada.

[8] T.F. Coleman, J.J. Moré (1983). Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20, 187–209.

[9] T.F. Coleman, J.J. Moré (1984). Estimation of sparse Hessian matrices and graph coloring problems. *Mathematical Programming*, 28, 243–270.

[10] S. Dalmas, N. Rostaing (1992). Automatic FORTRAN programs analysis and transformation using a typed functional language. In R. Glowinski (Editeur), *Proceedings of the 10th International Conference on Computing methods in applied sciences and engineering*, pages 527–535. Nova Science Publisher.

[11] R.S. Dembo, T. Steihaug (1983). Truncated-Newton algorithms for large-scale unconstrained optimization. *Mathematical Programming*, 26, 190–212.

[12] M. Desmadril (1990). *Programmer en C++, un langage orienté objet.* Eyrolles, Paris.

[13] S.I. Feldman, D.M. Gay, M.W. Maimone, N.L. Schryer (1990). A Fortran-to-C converter. Computing Science Technical Report 149, AT&T Bell Laboratories, Murray Hill, NJ 07974.

[14] H. Fischer (1991). Automatic differentiation of the vector that solves a parametric linear system. *Journal of Computational and Applied Mathematics*, 35, 169–184.

[15] H. Fischer (1991). Special problems in automatic differentiation. In A. Griewank, G. Corliss (Editeurs), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Proceedings in Applied Mathematics 53, pages 43–50. SIAM.

[16] M.R. Garey, D.S. Johnson (1979). *Computers and Intractability: a Guide to the Theory of NP-Completeness.* W.H. Freeman, New York.

[17] J.Ch. Gilbert (1991). Numerical methods for large-scale minimization. In *Proceedings of the 1991 ECMWF Seminar on "Numerical Methods in Atmospheric Models", Vol. 2, 9-13 September 1991*, pages 93–113. European Centre for Medium-Range Weather Forecasts, Reading, Berkshire RG2 9AX, England.

[18] J.Ch. Gilbert (1991). Non-linear optimization and large-scale problems. *Engineering Optimization*, 18, 5–21.

[19] J.Ch. Gilbert, J. Nocedal (1992). Global convergence properties of conjugate gradient methods for optimization. *SIAM Journal on Optimization*, 2, 21–42.

[20] G.H. Golub, C.F. Van Loan (1989). *Matrix Computations.* John Hopkins, London.

[21] A. Griewank (1989). On automatic differentiation. In M. Iri, K. Tanabe (Editeurs), *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, Dordrecht.

[22] A. Griewank (1991). The chain rule revisited in scientific computing. *SIAM News*, 24.

[23] A. Griewank (1991). Automatic evaluation of first and higher-derivative vectors. In R. Seydel, F.W. Schneider, T. Küpper, H. Troger (Editeurs), *Bifurcation and Chaos: Analysis, Algorithms, Applications*, 97, pages 135–148. Birkhäuser Verlag, Basel, Switzerland.

[24] A. Griewank (1991). Communication personnelle.

[25] A. Griewank (1992). Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1, 35–54.

[26] A. Griewank, D. Juedes, J. Srinivasan (1991). ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. Technical Report MCS-P180-1190, Argonne National Laboratory, Argonne, Il 60439.

[27] K.E. Hillstrom (1985). User guide for JAKEF. Technical Memorandum ANL/MCS TM-16, Argonne National Laboratory, Argonne, Il 60439.

[28] M. Hoshi, M. Iri, T. Tsuchiya (1988). Automatic computation of partial derivatives and rounding error estimates with applications to large-scale systems of nonlinear equations. *Journal of Computational and Applied Mathematics*, 24, 365–392.

[29] R.E. Huss (1990). An ADA library for automatic evaluation of derivatives. *Applied Mathematics and Computation*, 35, 103–123.

[30] M. Iri (1984). Simultaneous computation of functions, partial derivatives and estimates of rounding errors, complexity and practicality. *Japan Journal of Applied Mathematics*, 1, 223–252.

[31] M. Iri, K. Kubota (1987). Methods of fast automatic differentiation and applications. Research Memorandum RMI 87-02, Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo, Japan.

[32] M. Iri, K. Kubota (1990). PADRE 2, version 1 – User's manual. Research Memorandum RMI 90-01, Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo, Japan.

[33] M. Iri, T. Tsuchiya (1988). Analysis of rounding errors in large scale systems of nonlinear equations. *Proceedings of the Institute of Statistical Mathematics*, 36, 1–22. (English translation in: Research Memorandum RMI 89-02, Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo, Japan).

[34] N. Iwata (1984). *Automatization of the Computation of Partial Derivatives.* Master's thesis, Information Engineering, Graduate School, University of Tokyo.

[35] J. Joss (1976). *Algorithmisches Differenzieren.* Thèse de doctorat, ETH, Zurich, Suisse.

[36] L.V. Kantorovich (1957). On a mathematical symbolism convenient for performing machine calculations (en russe). *Dokl. Akad. Nauk SSSR*, 113, 738–741.

[37] G. Kedem (1977). Automatic differentiation of computer programs. In *Proc. 1977 Army Numerical Analysis and Computer Conference*, Madison, Wisc.

[38] G. Kedem (1980). Automatic differentiation of computer programs. *ACM Transactions on Mathematical Software*, 6, 150–165.

[39] K.V. Kim, Yu.E. Nesterov, B.V. Cherkasskiĭ (1984). An estimate of the effort in computing the gradient. *Soviet Math. Dokl.*, 29, 384–387.

[40] D.E. Knuth (1969). *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms.* Addison-Wesley, Reading, Massachusetts.

[41] J.L. Larson (1978). *Methods for Automatic Error Analysis of Numerical Algorithms.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL 61801-2932.

[42] J.L. Larson, M.E. Pasternak, J.A. Wisniewski (1983). Algorithm 594 – software for relative error analysis. *ACM Transactions on Mathematical Software*, 9, 125–130.

[43] J.L. Larson, A. Sameh (1978). Efficient calculation of the effects of roundoff errors. *ACM Transactions on Mathematical Software*, 4, 228–236.

[44] J.L. Larson, A. Sameh (1980). Algorithms for roundoff error analysis – a relative error approach. *Computing*, 24, 275–297.

[45] S. Linnainmaa (1976). Taylor expansion of the accumulated rounding error. *BIT*, 16, 146–160.

[46] J. Liu (1989). *Calcul formel et problème inverse.* Thèse de doctorat, Université de Paris IX, France.

[47] J. Liu (1989). GRADPACK: a symbolic system for automatic generation of numerical programs in parameter estimation. 5^{th} IFAC Symposium on Control of Distributed Parameter Systems, Perpignan, France, Juin 26-29.

[48] M. Metcalf, J. Reid (1989). *Fortran 8x Explained* (revised edition). Oxford Science Publications. Clarendon Press, Oxford.

[49] W. Miller (1975). Software for roundoff analysis. *ACM Transactions on Mathematical Software*, 1, 108–128.

[50] W. Miller, D. Spooner (1978). Software for roundoff analysis II. *ACM Transactions on Mathematical Software*, 4, 369–387.

[51] R.E. Moore (1981). *Methods and Applications of Interval Analysis.* SIAM Studies in Applied Mathematics 2. Society for Industrial and Applied Mathematics, Philadelphia.

[52] J. Morgenstern (1985). How to compute fast a function and all its derivatives, a variation on the theorem of Baur-Strassen. *SIGACT News*, 16, 60–62.

[53] S.G. Nash (1984). Newton-type minimization via the Lanczos method. *SIAM Journal on Numerical Analysis*, 21, 770–788.

[54] R.D. Neidinger (1989). An efficient method for the numerical evaluation of partial derivatives of arbitrary order. Technical report, Davidson College, Davidson, NC 28036.

[55] Yu.E. Nesterov (1991). Personal communication.

[56] D.P. O'Leary (1982). A discrete Newton algorithm for minimizing a function of many variables. *Mathematical Programming*, 23, 20–33.

[57] L.B. Rall (1981). *Automatic Differentiation, Techniques and Applications.* Lecture Notes in Computer Science 120. Springer-Verlag, Berlin.

[58] L.B. Rall (1984). Differentiation in PASCAL-SC: type GRADIENT. *ACM Transactions on Mathematical Software*, 10, 161–184.

[59] L.B. Rall (1987). Optimal implementation of differentiation arithmetic. In U. Kulisch (Editeur), *Computer Arithmetic, Scientific Computation and Programming Languages.* Teubner, Stuttgart.

[60] R.T. Rockafellar (1970). *Convex Analysis.* Princeton University Press, Princeton, New Jersey.

[61] J.W. Sawyer (1984). First partial differentiation by computer with an application to categorial data analysis. *The American Statistician*, 38, 300–308.

[62] L. Schwartz (1981). *Cours d'Analyse*, Tome I. Hermann, Paris.

[63] E.J. Soulié (1991). User's experience with FORTRAN compilers in least squares problems. In A. Griewank, G. Corliss (Editeurs), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Proceedings in Applied Mathematics 53, pages 297–306. SIAM.

[64] B. Speelpenning (1980). *Compiling fast partial derivatives of functions given by algorithms.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

[65] V. Strassen (1990). Alegebraic complexity theory. In J. van Leeuwen (Editeur), *Handbook of Theoretical Computer Science. Volume A: Algorithms and Complexity*, pages 633–672. Elsevier, Amsterdam.

[66] B. Stroustrup (1986). *The C++ Programming Language.* Addison-Wesley.

[67] O. Talagrand (1991). The use of adjoint equations in numerical modelling of the atmospheric circulation. In A. Griewank, G. Corliss (Editeurs), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Proceedings in Applied Mathematics 53, pages 169–180. SIAM.

[68] VAX UNIX MACSYMA: Reference manual, version 11 (1985). Symbolics.

[69] J. Vignes (1989). Optimal implementation of optimization methods and estimation of the accuracy of the results. In J.-P. Penot (Editeur), *New Methods in Optimization and their Industrial Uses*, International Series of Numerical Mathematics 87, pages 219–227. Birkhäuser Verlag, Basel.

[70] D.D. Warner (1975). A partial derivative generator. C.S. Technical Report 28, Bell Laboratories.

[71] R.E. Wengert (1964). A simple automatic derivative evaluation program. *Communication of the ACM*, 7, 463–464.

[72] J.H. Wilkinson (1965). *The Algebraic Eigenvalue Problem.* Oxford University Press, Oxford.